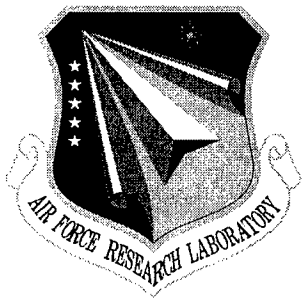
AFRL-IF-RS-TR-2001-59
Final Technical Report
April 2001



COOPERATIVE INTELLIGENT SYSTEMS FOR COMMUNICATION NETWORK MANAGEMENT

Clarkson University

Robert A. Meyer and Susan E. Conry

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK

This report has been reviewed by the Air Force Research Laboratory, Information Directorate, Public Affairs Office (IFOIPA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

AFRL-IF-RS-TR-2001-59 has been reviewed and is approved for publication.

APPROVED: Priscilla Cassidy

PRISCILLA A. CASSIDY
Project Engineer

FOR THE DIRECTOR:

WARREN H. DEBANY, Technical Advisor
Information Grid Division
Information Directorate

If your address has changed or if you wish to be removed from the Air Force Research Laboratory Rome Research Site mailing list, or if the addressee is no longer employed by your organization, please notify AFRL/IFGA, 525 Brooks Road, Rome, NY 13441-4505. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document require that it be returned.

REPORT DOCUMENTATION PAGE	Form Approved OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	APRIL 2001	Final Feb 91 - Apr 96

4. TITLE AND SUBTITLE
COOPERATIVE INTELLIGENT SYSTEMS FOR COMMUNICATION NETWORK MANAGEMENT

5. FUNDING NUMBERS
C - F30602-91-C-0029
PE - 33126F
PR - 2155
TA - 02
WU - 12

6. AUTHOR(S)
Robert A. Meyer and Susan E. Conry

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Clarkson University
Department of Electrical and Computer Engineering
Potsdam NY 13699

8. PERFORMING ORGANIZATION REPORT NUMBER
N/A

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Air Force Research Laboratory/IFGA
525 Brooks Road
Rome NY 13441-4505

10. SPONSORING/MONITORING AGENCY REPORT NUMBER
AFRL-IF-RS-TR-2001-59

11. SUPPLEMENTARY NOTES
Air Force Research Laboratory Project Engineer: Priscilla A. Cassidy/IFGA/(315) 330-1887

12a. DISTRIBUTION AVAILABILITY STATEMENT
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

This report documents new techniques for building intelligent, cooperative agents for assisting in the management of communication networks. First, we undertook a redesign and upgrade of an existing testbed for a distributed simulation environment. Second, we developed a generalization of previous work in distributed automated reasoning to provide for distributed expert systems built using an extension of conventional production rule technology. Third, we enhanced a distributed reasoning system by utilizing a stronger inference rule known as hyper-resolution. Fourth, we developed a distributed constraint based planner. Our results demonstrate a technique for solving the coordination problem among multiple problem solving agents in a distributed extension of the classical artificial intelligence blocks world domain. We have analyzed real world communication equipment, presented a typical problem scenario, and identified the expert knowledge required to solve the problem. We have experimental data that demonstrate significant improvements in the performance of distributed reasoning tasks. We also have experimental data that illustrate the effectiveness of our distributed constraint based planner. Finally, the problem of link activation scheduling is discussed. We show that our planner is able to solve this problem.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Intelligent Agents, Network Management	300
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

Abstract

This report documents a research project aimed at developing new techniques and methods for building intelligent, cooperative agents for assisting in the management of communication networks. The work has focused on four primary problem areas. First, we undertook a redesign and upgrade of an existing testbed for a distributed simulation environment. Second, we developed a generalization of previous work in distributed automated reasoning to provide for distributed expert systems built using an extension of conventional production rule technology. Third, we enhanced a distributed reasoning system by utilizing a stronger inference rule known as hyper-resolution. Fourth, we developed a distributed constraint based planner. Our results demonstrate a technique for solving the coordination problem among multiple problem solving agents in a distributed extension of the classical artificial intelligence blocks world domain. We have analyzed real world communication equipment, presented a typical problem scenario, and identified the expert knowledge required to solve the problem. We have experimental data that demonstrate significant improvements in the performance of distributed reasoning tasks. We also have experimental data that illustrate the effectiveness of our distributed constraint based planner. Finally, the problem of link activation scheduling is discussed. We show that our planner is able to solve this problem, and we have experimental results comparing the performance over varying communication network organizations.

Contents

List of Tables

List of Figures

Chapter 1

Overview

1.1 Statement of Work

1.1.1 Objective

Management and control of communication networks involves a combination of automatic controls with manual supervision and override capabilities. As the complexity of these networks has increased network management has come to rely on an increasing level of computer automation and assistance. The use of knowledge-based computer programs, such as expert systems, is becoming widespread in both commercial and military communication systems. A typical network management system planned for development and application in the 21st century will incorporate several distinct machine intelligent or artificial intelligent (AI) systems to assist humans in managing these complex networks. These AI systems will most likely be distributed among several nodes of a large, geographically dispersed network. These AI systems must also be able to solve problems which are distributed over functionally distinct problem areas corresponding to the variety of communications subsystems and multimedia networking expected to be available.

This research effort addresses the need for cooperative, machine intelligent aids which assist human operators who may become heavily overloaded in times of stress, or may not be aware of the scope of particular problems, or may simply lack experience in dealing with certain types of problems. In any of these situations these individuals could make effective use of an expert assistant to improve their response time in critical situations and to aid in generating high quality solutions to network management problems.

The objective of this project was to identify the significant design issues involved in the development of distributed AI technology and to show how this technology could be applied to communication network management. New techniques have been discovered which can enable effective cooperation of machine intelligent systems in solving the complex problems of network management. These techniques have been developed and tested in a variety of

application problems.

1.1.2 Background

Our previous work [62] on distributed problem solving for network management concentrated on problems in the context of networks having features similar to those found in the Defense Communication System (DCS) in western Europe. In that work we developed and implemented problem solving modules that perform such functions as distributed plan generation for service restoral, knowledge base management for systems involving multiple problem solvers, and construction of an underlying equipment and topological network knowledge base using a graphical user interface. In this new effort we have extended the scope of application to include modern digital networks and enhanced the functional capability to respond to dynamic network environments. New problem solving strategies were needed to meet these requirements.

1.1.3 Scope

This research project proposed the development of a testbed in which a communications network could be defined, simulated, and its management demonstrated. The proposed testbed would have extended an existing testbed developed as part of previous work. As will be discussed further under Section 1.2, the work on the testbed was modified after the first six month review. The major goal of the project was to investigate new techniques for cooperative intelligent problem solving. The most promising strategies were developed and demonstrated to illustrate applicability and feasibility. The results of this effort include a description of the significant design issues, alternative approaches considered, and test results for those designs implemented in the testbed.

1.1.3.1 Testbed Environment

The development of network management tools requires an appropriate testbed environment in which to test, evaluate, and demonstrate the performance of these products. The testbed serves two functions: (1) it provides a facility on which to perform the research studies necessary to address the design issues specified in the statement of work, and (2) it serves as a form of early prototype facility for demonstration purposes to assist the technology transfer process. This dual objective drives the testbed design, especially the user interface, toward one which is easy and natural to use and reflects as realistic an environment as possible. In order to convey complex network management tasks to military personnel in the field who lack complete training or experience in using these automated tools, the testbed environment must be able to communicate in an appropriate manner. A machine intelligent problem solving system should also demonstrate an intelligent operational environment and interface.

1.1.3.2 Distributed Problem Solving Agents

Based on our earlier work in distributed reasoning [56], we proposed to investigate methods for extending this work to a set of distributed expert agents. By far, one of the most perplexing obstacles faced by a set of distributed expert agents is the *coordination problem.* The coordination problem may be described as the dilemma faced by each agent in determining: 1) how to assess its role in the distributed problem solving process, and 2) how best to solicit assistance from other network agents so that its piece of the solution puzzle may be coherently integrated with theirs. A solution to the coordination problem relies heavily on characteristics of the particular problem domain and often becomes an integral part of the distributed problem itself. We proposed a technique which effectively reduces such coordination problems to equivalent problems in the domain of logic and relegates the responsibility for their solution to our already existing Distributed Automated Reasoning System (DARES). In effect, a coordination problem can be made *transparent* by exploiting the distributed problem solving heuristics previously developed for the DARES reasoning system. However, another distributed problem must be solved in its stead. The new problem is that of producing an adequate representation of the distributed problem in the form of an axiomatization which will bring about the desired coordination.

1.1.3.3 Distributed Reasoning

We proposed improvements to DARES, our distributed reasoning system [56]. DARES performs distributed reasoning by application of standard binary resolution techniques on a local basis, but with a modified control strategy. When a DARES problem solving agent perceives that it cannot make further progress or when it detects that it may be doing senseless work, the agent formulates a request for information and broadcasts this request to all other agents in the system. We proposed an alternative approach which would improve its performance through the use of hyper-resolution.

1.1.3.4 Distributed Constraint Based Planning

Problems associated with network reconstitution and restoral of service in the face of unanticipated network stress are very difficult to solve. The set of constraints that must be satisfied is complex, and it varies as the available assets change over time. Constraint satisfaction is, in general, an intractable problem requiring substantial computational effort in determining solutions to problems. The situation is only exacerbated when issues associated with a geographically distributed environment are introduced.

In order to accomplish the task of network reconstitution in a timely manner, it seems evident that each agent should be able to solve those parts of the problem that it can locally. Thus, when a problem is limited to one agent's area of responsibility, that agent should be able to handle the problem without involving any other agents. On the other hand, when

the problem and its solution must involve assets in other areas under the control of other agents, mechanisms for coordinating the activity of the group of agents are required. It is also clear that agents must be able to engage in problem solving at varying levels of abstraction, since no agent has a complete view of the global (non local) state. At any one time, an agent's view of the network is partial and may lack details available to other agents. Depending on the network state and resources available, an agent may be required to do the best job possible even with limited knowledge and without additional details.

Our previous work on plan generation and multistage negotiation [12, 14, 65] resulted in the development of a Distributed Multi Agent Planner (DMAP) and was done in the context of problems that arise in network reconstitution. We proposed to investigate distributed constraint based planning by taking a more general view of constraints, including a consideration of temporal constraints, in devising a general distributed constraint based planning system. Feasibility of the approach would be demonstrated through its application to problems in network management.

In this new effort we start with a "snapshot" of the network operating state and set of available assets. We then generate a set of candidate restoral plans in the form of a (distributed) set of control actions and negotiate among agents to arrive at a near-optimal set of actions that restore service to a maximal number of users. In devising the mechanisms and inter-agent communication protocols, we have utilized the concept of "resources" and constraints on resource availability to drive the search for a good solution to the problem.

1.2 Revisions to the Statement of Work

After the first six month review of the project, Rome Labs asked us to consider how this work might contribute to the CNOS II effort, then being developed by a commercial defense contractor. We were also told that the original plan for funding was to be changed with the result that the second year funding level was substantially reduced. The overall funding level would remain approximately as planned, but funds were shifted toward the end of the project.

We reviewed the CNOS II Exploratory System Model (ESM) software design and the Integrated Communication Network Management System (IMS). This review revealed strong similarities with our approach. We developed a new plan to build on the existing CNOS II technology and upgrade it to a fully supported environment.

The objective of this plan was to utilize the ESM software as the core component of a distributed testbed based on the architecture of our original SIMULACT [54, 53] system. The ESM software was developed by Stanford Telecommunications under the CNOS II program. It included many of the same ideas we used in building SIMULACT. The ESM, unlike SIMULACT, was specifically designed as a testbed for communication networks. As a result the user interface was more complete in those functions specific

to communications networks. It also included a Management Information Base (MIB). However, there were two serious limitations which would need to be overcome. First, the EMS user interface was built on the SunView graphic window system, an obsolete system no longer supported by Sun Microsystems, Inc. The ESM architecture was based on a fixed physical architecture with limited flexibility to accommodate a variety of network emulation/simulation configurations. In EMS each Sun workstation could serve as the local manager for one region of the network. An additional workstation was dedicated to the high level task of managing the overall simulation. Thus the level of complexity of a simulated communications network was directly limited by the number of workstations, even if there was spare compute power available.

1.3 Summary of Results

In this section we provide a high level summary of the results obtained in this project. As will be described below, we met some objectives while falling short in others. We believe the overall project was successful in terms of developing new techniques and algorithms for distributed problem solving. We were able to show direct application of these results to some areas of network management.

1.3.1 Testbed Development

Initial work on the testbed was aimed at developing major enhancements to our existing SIMULACT testbed and replacing its user interface system, GUS, with a new user interface system. We developed a User Interface Management System based on context-free grammars. This system would have been the basis for developing a communications network specific user interface to replace GUS. With the change in the statement of work as discussed previously, we did not continue with this development. Instead we undertook the process of understanding the design of the CNOS user interface software. We developed a complete design for DiST – a Distributed Simulation Tool. A key component of this design was the CNOS EMS software which had been built on the obsolete SunView window system. The process of converting the EMS software from SunView proved to be much more difficult and time consuming that we anticipated. The documentation available for the existing software was very minimal. Eventually, we were forced to abandon this part of the effort so as not to jeopardize successful completion of the remaining research objectives.

1.3.2 Distributed Problem Solving Agents

In this phase of the research we investigated new methods for finding solutions to problems where both domain knowledge and functional control are logically, and/or geographically

distributed. A distributed problem solving network is broadly defined as a distributed network of asynchronous, loosely-coupled, and semi-autonomous *agents* (processing elements) which cooperatively interact with one another to solve a *single* problem. Each agent is itself, a sophisticated problem solving system which has access to certain knowledge necessary to solve the problem. Portions of this knowledge may overlap arbitrarily with the knowledge accessible to other agents. However, no single agent has access to all the knowledge necessary; therefore no agent is capable of solving the problem by itself. Only by a cooperative exchange of knowledge among the agents can a solution to the problem be obtained. From the point of view of an individual agent, a good portion of any distributed problem involves the agent evaluating its own progress toward a solution, and determining how it can best coordinate with other agents to converge upon a globally coherent solution.

A new technique for solving distributed problems was developed in which an existing distributed automated reasoning system was used to facilitate interagent coordination. With this new technique, problem solving agents are programmed to be experts at reformulating distributed tasks as equivalent problems in the domain of mathematical logic. In doing so, the coordination problem associated with a distributed task is effectively transformed from one in the application domain into a corresponding problem in the automated reasoning domain.

We demonstrated how this technique can be applied in a classical AI problem solving domain: the Blocks World. The Blocks World is a simple domain involving a robot arm, a table, and a set of blocks. The objective is to have the robot develop a sequence of steps to bring the blocks into a desired configuration. We have extended this to a distributed domain involving multiple robots, tables, and blocks. While still a "toy" domain, the distributed blocks world incorporates many important features of a real world domain without the overhead of a very large store of real world knowlege.

We then developed TESS, an expert system shell written in the form of a production system. Unlike conventional production systems, TESS was specifically designed to operate in a distributed, cooperative manner and relies upon the DARES agents to conduct the distributed reasoning necessary to solve a problem with limited local knowledge.

The application of this approach to the domain of network management and control is then discussed in the context of the Defense Communication System. We completed a detailed analysis of actual communications equipment and the associated alarm monitoring that is commonly available. A problem scenario is then presented, and an expert's solution to this problem is illustrated. Based upon this, it then becomes a routine task of encoding the expert knowledge in TESS production rules.

1.3.3 Distributed Reasoning

We developed SHYRLI, a domain independent automated reasoning system. SHYRLI offers an improvement in performance over our previous domain independent reasoning systems that allows SHYRLI to be applied to "real world" problems. SHYRLI provides a baseline distributed reasoning case against which the performance of domain dependent heuristics can be measured.

The experiments with SHYRLI have given us insight into how distributed search can be different from single agent search. We have also discovered how SHYRLI's communication mechanisms allow us to identify information that can be kept "private" to a reasoning agent without affecting the systems progress towards a goal. SHYRLI has given us a view of how different distributions of knowledge can have very different affects on the performance of a distributed reasoning system.

Using SHYRLI we have developed a predicate calculus formalism that allows us to decompose a complex problem in to smaller ones. The formalism allows these smaller parts to be distributed over a distributed set of agents. We demonstrated the application of this paradigm using SHYRLI to the simulation of digital circuits.

To summarize the contributions of this part of our research, we have:

- Increased the performance of the architecture proposed in DARES by incorporating a stronger inference rule, hyper-resolution.
- Developed a distributed reasoning baseline from which different domain dependent coordination strategies can be compared.
- Added the concept of private knowledge to the DARES architecture.
- Added the concept of functional roles to the DARES architecture.
- Formulated a predicate calculus formalism that provides a means of distributing a complex task over a set of agents.
- Identified that different distributions of the same knowledge can have a significant effect on the performance of a distributed reasoning system.

1.3.4 Distributed Constraint Based Planning

We developed a new approach to distributed planning and implemented this design in a planner, DCONSA, that introduces agent autonomy in a cooperative planning context. This work built upon the expressive power of representational frameworks developed for multiagent planning combined with the coordination strategies found in distributed planning. We have made three primary contributions. The first is the development of a search

strategy in which agents exercise their autonomy through their individual guidance of a distributed localized search. The second contribution is a two stage planning process that exploits multiagent power by first planning for conjunctive goals in parallel and then seeking a satisfactory integration of these plans. Our work makes a third contribution by combining distributed planning with organizational self design. Previous work in distributed AI systems has shown that incorporating organizational self design can improve performance. By giving agents the capability to decide if and when to redistribute planning knowledge, they are able to dynamically reorganize the planning problem. Our experiments show that when such decisions are based upon a heuristic involving relative work load, the ability to reorganize improves planning performance.

The major features and contributions of the research we have conducted in the development of DCONSA are as follows:

- DCONSA is the first distributed planning system that incorporates agent autonomy into the planning process.
- DCONSA is the first planning system to perform searches for plans for conjunctive goals in parallel.
- DCONSA is the first distributed planning system to incorporate organizational self design.
- DCONSA's use of a formal distributed architecture allows us to describe a planning problem independent of the number of planning agents. This permits flexibility in the planning scenarios which can be modeled without changing the underlying problem definition.
- By selecting from among three modes of interaction, planning agents can determine how they will aid one another during plan construction. Since this decision is based upon relative work load, agents can dynamically balance the planning load in a distributed fashion.

1.3.5 Link Activation Scheduling

Our final result is an application of the DCONSA constraint based planner to a well known problem in modern multihop packet radio networks known as the link activation scheduling problem. This problem arises when some form of multiaccess channel is used, and a means for controlling access must be provided. In a distributed control environment, the access method must be implemented using a distributed control strategy.

We not only show that DCONSA is able to solve this problem, but we also demontstrate the impact of varying network organizations. These experiments illustrate the trade-offs between the advantages of parallelization in a distributed environment as compared to the costs of increased interconnectivity among the agents.

Chapter 2

Testbed Development

2.1 Introduction

Though natural language research in artificial intelligence has concentrated on spoken or written communication, it is clear that the traditional "natural languages" do not necessarily provide the most effective form of communication in domains such as communication network management. Much of the knowledge to be conveyed is graphical in nature, reflecting such network features as interconnection topology at various levels of the network hierarchy.

Our previous work resulted in the development of GUS and SIMULACT. GUS [39] is a graphical user interface that permits the user to describe various network components and their interconnection in a natural, graphical fashion. Network topology can be viewed at various levels in a flexible way, and the underlying representation that is built is one that the problem solving modules can utilize effectively. The network model is one that is closely related to the structure of the current DCS. Thus GUS interacts with the user using a graphical interface and builds a frame based representation that problem solvers can use efficiently.

SIMULACT [54, 53] is a development environment for distributed problem solving systems. It provides a computational platform for an applications programmer to specify and implement problem solving modules in a distributed environment. SIMULACT incorporates interactive monitoring and debugging facilities, as well as a diary facility that makes post-mortem analysis feasible.

We proposed to extend this work by developing an interactive user interface to a communications system domain problem solver that is based on a more generic model and incorporates the following features:

1. the ability to define instances of generic communications equipment objects as well as new object types and their properties,

2. an interface with an external SQL-based network database,

3. an interactive graphical user interface that permits multilevel views of the network, views from multiple perspectives, and the ability to monitor problem solving activity on an interactive basis

4. an intelligent front end that permits hypothetical reasoning, so that speculative network analysis can be supported

We first discuss a User Interface Management System we developed to use in building a user interface which could meet the above objectives. As we were completing this task, the statement of work was modified as described previously in Section 1.2. As a result of this modification, we directed our efforts in the testbed development toward the design of a completely new Distributed System Testbed (DiST). The design of DiST is presented in the second section of this chapter.

2.2 User Interface Management System

In order to meet the goals for an enhanced user interface, we conceived a User Interface Management System (UIMS). The UIMS provides a method for easily creating and maintaining user interfaces. It does this by providing a set of standard parts and the tools for putting these parts together. Our User Interface Management System is based on a subset of the Context-Free Grammars (CFG) known as S-grammars and a subset of the Attributed Translation Grammars. In addition to the grammar, the user interface uses a set of graphical objects, which provide graphical output to the display and input to the parser. The UIMS has been made portable across different underlying hardware/software platforms by encapsulating the interface in the input and output systems.

Several different UIMS models exist. The most popular models are the event handler model [35], transition diagrams [42], and context-free grammars [64]. Our UIMS is based on a modified context-free grammar and employs an object-oriented approach in a Common LISP Object System environment. Although simple, it has been used to develop user interfaces for a variety of different applications.

2.2.1 Overview of the UIMS Architecture

Figure 2.1 presents the architecture for a prototypical user interface developed using this UIMS. As can be seen in the figure, the user interface is divided into four parts: the user interface specification and parser, the graphical objects, the input system, and the output system. The main reasons for dividing it into these four parts are modularity and encapsulation.

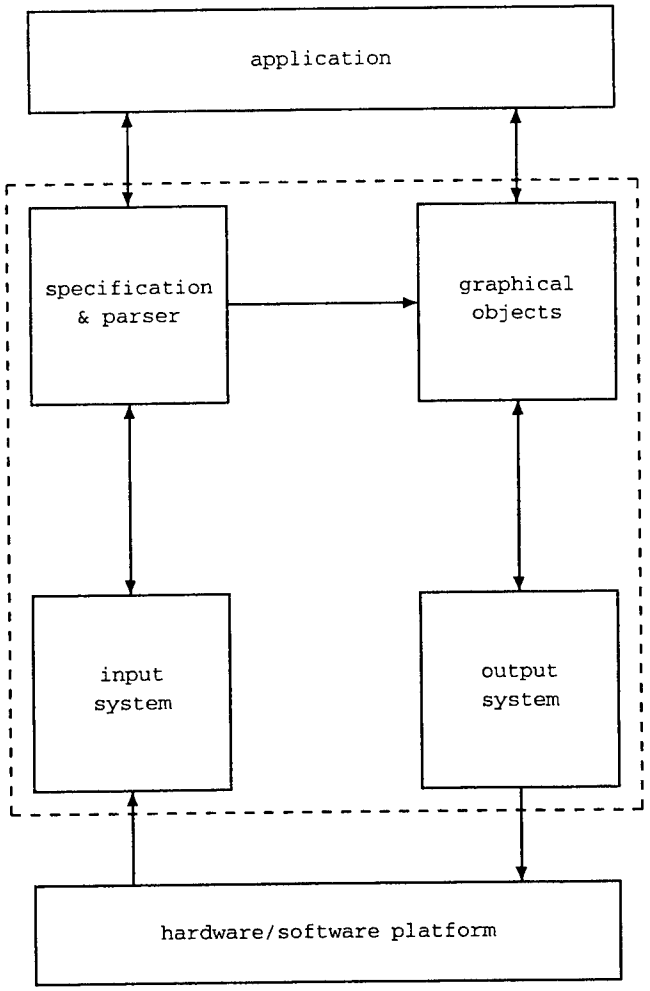


Figure 2.1: User Interface Architecture

The specification and parser provide the flow of control for the user interface, and will be discussed in more detail later. The set of graphical objects is an object-oriented class hierarchy of objects used to provide graphical output. Objects that are *mouse-sensitive* are also used as a source of user input to the parser; they contain a *type* that represents the token given to the parser when the object is mouse-clicked by the user. The UIMS contains a library of predefined standard graphical object classes. A user-interface developer may use these existing classes or define new object classes for his objects. The input and output systems provide a portable method of performing I/O with the underlying hardware/ software platform.

In order to define a user interface with this UIMS, the developer needs to define the user interface specification representing the dialog. Part of this specification includes embedded action routines which call application specific functions, also defined by the developer. These

functions may make use of graphical objects from the library of standard objects, or objects defined by the developer specifically for this application.

2.2.2 Modified Context-Free Grammar

A CFG-based specification was chosen for our UIMS, rather than state transition diagrams, or event handlers [42, 35]. In order to create a deterministic pushdown-automaton (PDA) from the specification, a restricted form of CFG was used — the S-grammar [52]. A Modified Context-Free Grammar (M-CFG) is a combination of an S-grammar and an attributed translation grammar [51]. We define a M-CFG as follows.

> An M-CFG is a set of terminal symbols, T, a set of non-terminal symbols, N, a set of function-codes, F, a set of mouse-documentation strings, M, a set of productions, P, and a starting symbol, S. Each symbol in T and N, and each function-code in F, has an associated attribute that may take on any value. Each member of P must be of the form $(A \rightarrow a\ \beta\ m)$ such that $A \in N$, $a \in T$, $\beta \in (N + T + F)^*$, and $m \in (M + \epsilon)$. Further, if $A \in N$, $a, b \in T$, $\alpha, \beta \in (N + T + F)^*(M + \epsilon)$, and $(A \rightarrow a\ \alpha)$, $(A \rightarrow b\ \beta) \in P$, then $a \neq b$.

Using an S-grammar simplifies parsing: at any non-terminal state, the particular production to use can be determined by the next input token received. No lookahead token is required to choose the next production. This also allows prompting to be achieved easily.

2.2.3 Example User Interface Specification

The following example is a simple dialog box presented to the user if he tries to quit the application without saving an open file. This dialog box could be represented by a small grammar, which simply asks the user to choose which action to follow.

Three graphical objects are required for this small example — one pushbutton for each of the three options presented to the user. The first part of the example creates three graphical objects that represent the pushbuttons; each pushbutton has a label and a type. This type is used as input to the parser.

Following the button definitions is the user-interface specification. Note that the terminal symbols are given by the type of the graphical object and the mouse button that the user clicked. When the specification is parsed, the user has three choices — he may press either the *yes*, *no*, or *cancel* button, at which point the associated embedded action routine will execute. Also note the prompt information stored as the last item of each production. In this example, if the mouse pointer is *over* the yes button, the user will see the "Yes. Save file." prompt. Moving the mouse over the other buttons will present their respective prompts.

```
(defvar *yes-button*    (make-a-simple-button "Yes"    :yes-button))
(defvar *no-button*     (make-a-simple-button "No"     :no-button))
(defvar *cancel-button* (make-a-simple-button "Cancel" :cancel-button))

(defvar *parser*
  (create-parser '((:terminals :yes-button%mouse-1 :no-button%mouse-1
                               :cancel-button%mouse-1)
                   (:non-terminals S)
                   (:start-symbol S)
                   (S -> :yes-button%mouse-1 ( (save-file) (exit) )
                         "Yes. Save file.")
                   (S -> :no-button%mouse-1 ( (exit) )
                         "No. Don't save file.")
                   (S -> :cancel-button%mouse-1 ( (cancel) )
                         "Cancel. Don't quit."))))
```

In order to complete this example, the developer must define the application-specific routines `save-file`, `exit`, and `cancel`. The developer must also define the button graphical objects as mouse-sensitive objects. To invoke this dialog box, the application must display the relevant graphical objects (the three buttons), and start the parser. When the parser returns control to the application, the user will have selected an action, and the corresponding routine will have been executed.

This UIMS has been used to create user interfaces much more complicated than the simple example presented here. This example is intended to illustrate the fundamental design features. As was discussed in Section 1.3.1, after the UIMS had been developed, but before it could be used to build the user interface for the testbed, the statement of work was revised so as eliminate further development along these lines.

2.3 DiST: A Distributed System Testbed

The design of DiST was based upon the requirement that we incorporate the core software from the CNOS II Exploratory System Model (ESM) and the Integrated Communication Management System (IMS). Our analysis of the design documents revealed several areas of similarity between the ESM/IMS and our SIMULACT. There were, however, significant areas of difference. For example, SIMULACT (and the problem solving agents which ran under SIMULACT) was written in Common Lisp. The ESM and IMS were written in the C programming language. The ESM incorporated a more complete, graphical interface, but one less capable than we had intended as the enhancement for SIMULACT. The ESM architecture was based on a fixed physical architecture with limited flexibility to accomodate a variety of network emulation/simulation configurations. In particular, in the ESM/IMS software, each workstation is assigned the role of exactly one network management node. Another workstation is used for the Executive Director function to oversee the entire simulation. For DiST we envisioned a logical architecture in which a workstation could be

assigned one or more management nodes. With this approach, a fixed set of workstations could simulate networks having a wide range of management nodes.

After the initial design of DiST was complete, we began the implementation task. The intended plan was to build on the existing code for the ESM/IMS. At this point we began to encounter a series of problems which eventually became insurmountable. The ESM/IMS code relied on SunView for all of its user interface. SunView was an obsolete product of Sun Microsystems for which there was no current support available. Also, the only documentation for ESM/IMS was the code itself. After a significant investment of time and effort, we concluded that our prospects for using the ESM/IMS code were not good. Since work on the other components of the overall project needed to progress with some knowledge of the simulation environment, we decided to fall back on our original testbed system. It was our belief that the research goals for distributed problem solving could be demonstrated using SIMULACT, and it was more important to spend the time and effort on achieving those goals than producing an enhanced testbed.

Chapter 3

Distributed Problem Solving Agents

3.1 Introduction

3.1.1 The Coordination Problem

Since its inception, the field of study generally known as Artificial Intelligence has expanded to a point where it now includes any area of research which strives to achieve "intelligent" behavior in the class of systems under its consideration. Distributed Problem Solving (DPS) is one such research area and will be the setting for the material presented in this report. Briefly described, research in distributed problem solving is primarily concerned with finding solutions to a *single* problem whose domain is logically and/or geographically distributed. A prevalent representational framework for such problems assumes that knowledge of their underlying distributed nature is captured by a network of problem solving *nodes*. The knowledge represented by each network node corresponds to some convenient partitioning of the problem space. Usually this partitioning will coincide with the inherent logical and/or geographical distribution. The component of each network node responsible for local problem solving activities, consists of an asynchronous computing element generically known as an *agent* (or *problem solver*). The distribution of knowledge throughout the network is assumed to be such that no single agent is capable of solving the distributed problem all by itself. Rather, the network of problem solving agents must cooperate with one another in order to coordinate their individual efforts and collectively formulate a solution to the overall distributed problem.

Distributed problem solving agents communicate with one another using a message passing paradigm. Control of the problem solving process is exerted through various exchanges of information among the agents. In this work inter-agent communications are designed to be *loosely coupled*. Tasks are said to be coupled when the input of one task depends on

the output of another. A coupling of tasks is considered to be "tight" when a state change in one task immediately affects the state of another task (i.e. task synchronization). Slack resources can be introduced, in the form of buffer inventories, to "loosen" the coupling between tasks. As one task places resources (information, partial products, etc.) into a buffer, coupled tasks can remove them as they are needed. In this way, vacillations in a task's output productivity will be absorbed by the buffer and thus will not directly affect the processing of coupled tasks. With respect to distributed problem solving systems, slack resources are used to reduce coordination complexity by buffering inter-agent communications with message queues. Information communicated from one agent to another is placed into the receiving agent's message queue and is attended to when time permits.

In many problem domains, the cost of inter-agent communication can be prohibitively high or the communication channels themselves may be near their capacity. Therefore, communication between problem solving agents should be kept to a minimum. This is especially true of problem domains which are geographically distributed. In such domains, any indiscriminate inter-agent communications may cause undue contention for the communication channels and tend to hinder the overall problem solving effort. A loosely coupled architecture would alleviate this contention by requiring that an agent spend the "majority" of its time engaged in local problem solving activities with relatively little time spent initiating and responding to inter-agent messages. It is not permissible however, for an agent to become loosely coupled to such an extent that it simply ignores messages sent to it by other agents. A second condition imposed on the distributed agents effectively prevents such a situation from ever occurring. This condition maintains that agents be *semi-autonomous.* A semi-autonomous agent is one which is required to process requests for knowledge from other agents in a timely manner as well as comply with any instructions it may receive, but is otherwise free to pursue its own problem solving agenda.

The use of *slack resources* is another organizational substructure capable of reducing coordination complexity. Another complexity reducing substructure concerns the way in which an organization is partitioned. Organization theory distinguishes between two types of organizational partitioning called self-contained division and functional division. In an organization which is made up of self-contained divisions, each division contains all the necessary production facilities to produce a different product. Conversely, in functionally divided organizations, different products are produced by sharing production facilities, each of which performs a single function. With respect to a distributed processing system, partitioning an organization into self-contained divisions corresponds to having a group of equally capable homogeneous problem solving agents; functional divisions correspond to a group of highly specialized agents. Depending on the nature of the distributed problems under consideration, one type of partitioning will reduce complexity while the other will increase complexity.

By far, the most perplexing obstacle in solving a distributed problem under this paradigm is the *coordination problem.* The coordination problem may be described as the dilemma

faced by each agent in determining; 1) how to assess its role in the distributed problem solving process, and 2) how best to solicit assistance from other network agents so that its piece of the solution puzzle may be coherently integrated with theirs. A solution to the coordination problem relies heavily on characteristics of the particular problem domain and often becomes an integral part of the distributed problem itself. A technique is introduced which effectively reduces such coordination problems to equivalent problems in the domain of logic and relegates the responsibility for their solution to an already existing Distributed Automated Reasoning System called DARES [56]. In effect, a coordination problem can be made *transparent* by exploiting the distributed problem solving heuristics previously developed for the DARES reasoning system. However, another distributed problem must be solved in its stead. The new problem is that of producing an adequate representation of the distributed problem in the form of an axiomatization which will bring about the desired coordination. An example development of such an axiomatization for problems in a toy domain is presented in detail and the potential for an expert system implementation is discussed. A small example taken from the domain of a large-scale communication system is presented as further evidence of the automated reasoning technique's usefulness.

3.1.2 Related Work

3.1.2.1 Distributed System Organization

The organization of a distributed system is considered to be a composite of both an (organizational) structure and control regime. An organizational structure establishes a suitable framework within which all aspects of the distributed problem solving process are to be carried out. Its very nature represents a structural compromise between opposing organizational forces. These forces are driven by the need to accommodate certain troublesome attributes of distributed problems and their domains. A control regime consists of machinations which are intended to bring about an overall coordination of distributed problem solving activity. The initiatives undertaken by an agent must be coordinated with those of its partners in order to ensure each agent will produce the proper resources (services, partial products, etc.) at the proper time. Possible strategies for the realization of such coordinated problem solving behavior are constrained by the functional properties of an organization. Thus, it becomes apparent that what constitutes a practical mode of coordination is dependent on a wide variety of problem characteristics and how they are accommodated by an organization's structure and control regime. To date, no well-defined methodologies for synthesizing an appropriate coordination stratagem have yet been developed. However, many useful insights into their design can be obtained by considering previous work [27] in which distributed systems are viewed as being analogous to human organizations.[1] It has been found that many of the concepts and theories germane to the management science field of

[1] A brief introduction to the advantages and disadvantages of distributed systems and the need to reduce uncertainty and complexity can also be found in [7].

organization theory can also be applied to distributed processing systems. These concepts are summarized in the following paragraphs and will serve as a basis for the discussion of several distributed processing organizations and the coordination strategies which have been developed for them.

Not unlike the human mind, the capacities of distributed agents for problem solving are limited both by their rate of program execution and effective use of available memory. These limitations result in a condition which is known as *bounded rationality.* With respect to a distributed organization, bounded rationality has two important implications; 1) there is an imposed upper bound on the amount of information an agent can process effectively, and 2) the extent to which an agent may exert control within the organization is limited. As distributed problems become increasingly larger in size and more diverse, the amount of information which must be processed also increases, as does the degree of control complexity. Eventually, the requirements of ever more difficult and involved distributed problems will begin to tax, and even exceed, the maximum computational capacity of a given organization. When this condition occurs, the organization is ultimately destined to fail. A major goal in the design of distributed organizations is therefore to provide an effective means of limiting the amount of information and control complexity perceived by any member of an organization. For any given distributed problem, a successful organization will prevent an agent from becoming inundated with information and control decisions.

The structure of an organization depends on the degrees of uncertainty and complexity associated with tasks in a particular problem domain. Uncertainty is defined as the difference between available information and the information necessary to make the best possible decisions, and complexity is defined as the degree to which excessive demands are placed on the rationality of an organization. Organizational design strategies suggested by contingency theory are based on the manifestation of uncertain information. In a departure from this view, [27] identifies three additional types of uncertainty (algorithm, environment, and behavior) which can generally be ascribed to distributed processing systems. In addition, three types of complexity (information, task, and coordination) are also distinguished. For tasks which exhibit a high degree of complexity, organizations having a heterarchical structure in which necessary resources are obtained through *contracting* are generally more desirable. While for tasks which exhibit a high degree of uncertainty, it is advantageous to "vertically integrate" facilities to produce the necessary resources into a more hierarchical structure. The selection of an appropriate organizational structure for a distributed problem domain depends largely on the nature of its tasks, and will likely fall somewhere between the two extremes of being a heterarchy or hierarchy.

An agent may have reasons, based on knowledge of the problem domain or the nature of specific tasks, to doubt the correctness of the available information. This is called information uncertainty. Organizations can reduce information uncertainty by incorporating control mechanisms which use synthesis or predication techniques to verify intermediate problem solving results. The algorithms used by an agent to make various control decisions

are also a source of uncertainty. Regardless of the reliability of information upon which control decisions are based, it is uncertain whether these decisions will bring about the intended environmental state changes. The interaction of many concurrent control decisions has a tendency to obscure the coordination objectives of any one particular agent. In this respect, no agent can be certain of what environmental state will result from such interactions. This condition is what is called environmental uncertainty. In the face of environmental uncertainty, an organization must have the capability of adapting to any unforeseen problem solving states. An organization can reduce algorithm uncertainty by having agents maintain multiple approaches to fulfilling the requirements of a task. In this way, the organization can recover from poor control decisions by employing an integration technique such as relaxation. The last form of uncertainty is concerned with the behavior of problem solving agents. Behavioral uncertainty makes it necessary an agent to consider contingency plans in the event another agent is unable to provide the resources it has "promised". This form of uncertainty is primarily associated with heterarchical types of organizations.

The sheer quantity of information which agents are exposed to during the course of normal problem solving activities, is measured as information complexity. If an agent must contend with large amounts of unrefined information, the agent's rationality can quickly become over burdened. When such a condition occurs, the agent is no longer an effective member of the distributed problem solving group. One method by which the level of information complexity can be reduced is through information abstraction. Abstracted information provides several levels of representation in which an agent may choose to carry out its processing activities. The lowest level of representation contains the highly detailed unrefined forms of information while successively higher levels contain forms of information which are increasingly more general in nature. As the levels of abstraction get progressively higher, the extent to which detailed information is concealed continually increases. Using abstracted information, an agent can reduce its processing burden by selecting the highest levels of representation possible in which to carry out its problem solving activities.

Task complexity is a measure of the total number of actions which must be performed to complete a task. The complexity of a particular task can vary significantly depending on the organizational structure in which the task is to be performed. In a strict hierarchical organization, the level of required communication among agents (i.e. coordination complexity) increases as tasks become increasingly more complex. As task complexity continues to increase, a level will eventually be reached in which effective problem solving is no longer possible due to an extreme amount of communication overhead. The requirements of tasks at this level of complexity have exceeded the bounded rationality of the hierarchical organization and must be reduced. An effective reduction in task complexity can be realized by moving to a heterarchical type of organizational structure. In a heterarchy, resources needed by an agent to complete a task may be obtained from other agents by a method known as *contracting*. Once an agent has made its need for a certain resource known, another agent may decide it can supply the resource and agree to do so. This agreement forms

a contract between the two agents and eliminates the need for any further interactions until the resource is subsequently tendered. Contracting therefore is capable of effectuating considerable reductions in the amount of communication overhead which in turn translates to a reduction in task complexity. This is the basic premise of the division of labor principle.

As may already be evident, coordination complexity and task complexity are not independent, but rather are quite strongly linked. Resource interdependence among problem solving agents makes it necessary for the agents to cooperate with one another in order to complete a task. Such cooperative interactions between agents must be coordinated so that essential resources are made available and properly exchanged when they are needed. The amount of agent interactions necessary to accomplish a task coincides with the level of coordination complexity. There are few heuristics available to use as a guide in determining how to reduce the complexity of task coordination. Perhaps the most widely used heuristic stems from the definition of *nearly decomposable* systems. A system is considered to be nearly decomposable if it exhibits a greater level of interaction within a problem solving agent (interaction locality) than between agents. This implies that coordination complexity is directly related to how well the decomposition of a task exhibits interaction locality. To the extent a task decomposition manifests interaction locality, greater reductions in coordination complexity will be realized. As a design heuristic for distributed organizations, system near decomposability implies they should be constructed so as to encourage minimal interactions between problem solving agents.

3.1.2.2 Distributed Control Regimes

The agents of a distributed problem solving system cooperatively interact with one another through the transmission of various kinds of messages over a network of communication pathways. A coordination of these interactions is necessary to bring about acceptable and globally coherent problem solving behavior. However, coordination of the problem solving process as a whole cannot be achieved directly because there is no centralized form of network control. The only available alternative is a form of decentralized control in which agents must determine for themselves how best to interact with one another. From a communications perspective, a decentralized control regime can be characterized with respect to the assumptions made locally by agents concerning the nature of interagent coordination. Principally these assumptions pertain to who is going to receive messages sent by the agent, how are these messages going to be processed, who will send the agent messages, what kinds of information will these messages contain, what processing does the sending agent expect to be done, and what response will the sending agent expect to receive. A spectrum of decentralized control regimes can be discerned by considering the relative level of control present in interagent coordination. The level of control used to coordinate agent interactions can be described in terms of both the specificity with which the character of tasks are conveyed and the extent to which these tasks are expected to be carried out.

With regards to the specificity of received messages, those which specify *tasks* to perform are considered more specific than those which specify *goals* to achieve. Likewise, messages which specify goals to achieve are considered more specific than those containing only *data.* Agents which are required (to some degree) to perform certain actions in response to the receipt of a message, are generally referred to as being *externally directed.* Whereas those agents which decide (to some degree) for themselves how they will respond to received messages are referred to as being *self-directed.* Located at the two extremes of such a control regime spectrum are complementary forms of decentralized control. At one extreme are forms of control which are classified as being "implicit" while those located at the other extreme are classified as being "explicit". In any given control regime, the more externally directed an agent and the more specific a received message, the more explicit the form of control. In contrast to the explicit forms of control, the more self- directed an agent and the more general a received message, the more implicit is the form of control. It is easy to imagine a whole array of control regimes in which their level of control would place their classification somewhere between the two extremes of being an implicit or explicit form of control.

3.1.2.3 Distributed Organizational Structures

The design of conventional distributed problem solving systems can be viewed as a decomposition of what are essentially centralized systems. In a centralized system, the knowledge contained in their databases are assumed to both complete and accurate. For such centralized problems, the knowledge contained in a system's database is both complete, meaning there is sufficient knowledge to solve the problem, and accurate, meaning all knowledge is valid. Such systems are generally organized so as to provide each agent with a consistent local knowledge base containing only a portion of the overall domain knowledge. When taken as a whole, the distribution of knowledge throughout the problem solving network represents a complete and accurate collection of all available domain knowledge.

The design of conventional distributed systems inherently places a great deal of emphasis on maintaining correctness in every aspect of the problem solving process. Since the knowledge distribution is known to be accurate, problem solving agents can be certain that any result they derive from a valid computation is indeed correct. Similarly, any result which is obtained by an agent through a cooperative exchange with another, must also be correct and may readily be incorporated into the agent's local problem solving effort. Such conventional distributed systems have been classified in [49] as being of a *completely accurate, nearly autonomous* (CA/NA) type.

For applications of distributed problem solving to areas in which the domain knowledge itself is neither complete nor consistent, the use of a CA/NA type of distributed system would not be appropriate. Another kind of distributed system called *functionally accurate/cooperative* (FA/C) [49] has been defined which is able to cope with domains having

these characteristics. Rather than constraining agents to producing only results which are correct and complete as is done with CA/NA systems, the FA/C approach permits agents to generate tentative results which may be incorrect, incomplete, and insistent with the tentative results produced by other agents. In this way, an agent can contend with incomplete domain knowledge by generating alternative results based on plausible expectations of what the missing knowledge would have perhaps supported. An interactive process of *iterative refinement* can then be used to cooperatively eliminate any erroneous intermediate results.

By relaxing the constraint of CA/NA distributed systems which required an agent to generate only complete and correct results, agents faced with the prospect of possibly incomplete domain knowledge can still generate some useful information. An agent can produce a set of alternative partial results which are based on reasonable expectations of what the missing knowledge may be. These tentative results may be incomplete, incorrect, and inconsistent with the tentative results produced by other agents.

Another source of information uncertainty not distinguished by Fox, but which was recognized by Lesser and Corkill [49] is the possibility of incomplete information resulting from a limited distribution of domain knowledge. In such distributions, there is certain information missing from the overall problem description which can never be explicitly attained by any computational means.

3.1.2.4 Coordination Strategies

It has generally been recognized there are two basic forms of cooperation in DPS systems called *task sharing* and *result sharing*. Task sharing is a form of cooperation in which problem solving agents assist one another by sharing the responsibility for solving subtasks associated with the overall distributed problem. In contrast, with result sharing agents cooperate with one another by supplying partial results which may be incorportated into an agent's local problem solving effort. There are several reasons why it would be advantageous (if not necessary) for an agent to decompose a large task into a number of smaller more manageable subtasks. Determining how subtasks should be delegated to other network agents is commonly called the *connection* problem. As a rule, it can be said that "optimal" problem solving performance may be achieved through a balance of computational loading among the network of distributed agents. For this reason, an agent endeavoring to solve a task which is too complex may cause a computational "bottleneck" that would hinder the overall problem solving effort. In such cases, an agent's time would be better spent partitioning a difficult task into a number of constituent subtasks. The subtasks would in turn be passed along to other agents of the network which are better able to accommodate the additional load needed to compute a solution. Alternatively, an agent may find it necessary to decompose a task, not necessarily as a result of it being too complex, but rather because solving the task requires certain expertise which the agent simply cannot provide. Subtasks of this kind must be delegated to other agents on the basis of their

abilities as well as their relative computational load.

Negotiations initiated between a contract manager and various subcontractors in distributing the labor required to complete a task, has been suggested as a metaphor for describing inter-agent cooperation. The Contract Net formalism [76, 77] is a protocol which is used to facilitate negotiations between pairs of DPS agents. A contract is considered an explicit agreement between an agent (contract manager) which has generated a subtask to be completed and an agent (subcontractor) which is charged with completing the task. The role of an individual agent as being either a contract manager or subcontractor is not chosen *a priori,* but rather is determined naturally as the problem solving process unfolds. To establish contracts, managing agents broadcast messages to the other distributed agents in the form of task announcements. Each available agent (potential subcontractor) evaluates the task announcements it has received in order to assess; 1) the relative difficulty of each task, and 2) how appropriate the agent's expertise is for the particular task. Based on such evaluations, an agent will submit a bid on those tasks for which it is best suited. The managing agent then evaluates these bids and awards contracts to the most appropriate agents charging them with the completion of each subtask. Contract awards serve to establish connections between corresponding managers and subcontractors in which they may communicate privately while work on the contract is in progress. As the contracted work for each subtask is completed, the managing agent is responsible for integrating these results into its solution of the original task.

There is no restriction which prevents a subcontractor from further decomposing a task and thus becoming the managing agent for those respective subtasks. In this way, the Contract Net formalism provides a dynamic method of recursively defining a distributed control hierarchy. The essence of such a distributed control structure is to provide a framework for coordination in which partial solutions of the distributed problem can be mechanically integrated into a globally coherent solution. This method of coordinating distributed agents is particularly appealing since the synthesis of a solution exactly parallels the structure of the established control hierarchy. A more detailed discussion of Contract Nets and issues concerning their implementation are presented in [18].

3.2 Distributed Task Coordination With Automated Reasoning

3.2.1 An Overview of DARES

DARES [57, 59] is a distributed automated reasoning system which was developed as a vehicle for investigating the role of knowledge in distributed problem solving systems. DARES utilizes a refutation based theorem proving technique similar to that of many classical automated reasoning systems. What sets DARES apart from other automated reasoning

systems is its inherently distributed architecture. In a conventional automated reasoning system, a reasoning problem is solved by a single theorem proving process which has access to all the relevant domain knowledge. In contrast, the DARES system is able to solve the very same reasoning problems when the domain knowledge is distributed among a network of identically configured reasoning agents. The distribution of knowledge among these reasoning agents is such that no single agent may have enough knowledge that would enable it to complete the reasoning task alone. This is a characteristic of distributed problem solving environments and requires the reasoning agents to cooperate with one another in order to solve a given reasoning problem.

Each reasoning agent is configured as the manager of a set of independent and concurrent theorem prover processes, each of which performs binary resolution using a set of support strategy. Individual theorem provers under the control of a single reasoning agent are dedicated to solving completely separate reasoning problems. To solve a reasoning problem, the task must first by relegated to a subset of the reasoning agents in the network. When a subset of reasoning agents has been selected, a unique tag is associated with the problem and the selected agents are notified of their participation in that particular reasoning task. The notified agents then allocate a new theorem prover process which is identified by the chosen theorem tag. In this way, every reasoning agent working on a particular problem has a theorem prover process identified by the same theorem tag. This allows a reasoning agent to handle communications from other agents concerning a particular reasoning task. For purposes of experimentation, any knowledge relevant to the problem is distributed among the subset of selected agents. In practice however, it is expected such knowledge is already distributed and can be made available. The resolution process then begins.

At the completion of each resolution level, the individual DARES reasoning agents apply a *Forward Progress* heuristic to determine whether or not they have made any significant progress towards a solution. The Forward Progress heuristic is comprised of two other heuristics, called the *Proof Advancing* heuristic and the *Monotonic Search* heuristic, both of which analyze the newly generated resolvents for apparent signs of progress. The Proof Advancing heuristic compares the length of each resolvent, in terms of the number of literals it contains, with the lengths of its parent clauses. Generally, one can expect resolution to generate resolvents which have a length two less than the sum of the lengths of their parent clauses. Resolvents which fit into this category offer no evidence the proof is actually advancing. On the other hand, resolvents which have a length less than what may be expected, are considered to represent an advancement in the proof. A resolvent with less than the expected number of literals will be generated when a unified substitution into a parent clause results in a number of identical literals. All but one of these identical literals is redundant and will be removed from the resolvent clause, thus producing a shorter resolvent.

The Proof Advancing heuristic also recognizes two important special cases in which the generated resolvents do have the expected number of literals. The first case is a resolvent clause which has only a single literal, or *unit clause.* When resolving with a unit clause,

any parent clause resolved with the unit clause will generate a resolvent with one less literal then the parent clause. It is for this reason that unit clauses are viewed as being capable of narrowing the search and are therefore considered as advancing the proof. This same idea is used in the *Unit Preference* resolution strategy where clauses with the smallest number of literals are resolved before those clauses containing more literals. Those resolvents that were generated from a unit clause are the second special case recognized by the Proof Advancing heuristic. They represent one of the first steps in narrowing the search from a unit clause as mentioned above. If there is at least one resolvent that fits into any of these three categories, the Proof Advancing heuristic is satisfied and the reasoning agent continues its search by generating the next level of resolvents.

The Monotonic Search heuristic takes a somewhat broader view of what is considered progress than does the Proof Advancement heuristic. Rather than considering only the number of literals in each clause at a particular level of resolution, the Monotonic Search heuristic looks at the *total* number of literals in all the clauses. For the search of a proof to be considered *monotonic,* the total number of literals contained within the clauses of a particular resolution level must be less than the total number of literals in the preceding level of resolution. Having stated this, the operation of the Forward Progress heuristic can now be summarized as follows: if the Proof Advancing heuristic is not satisfied by the current level of resolution and was also not satisfied by the *previous* level of resolution, and the two resolution levels do not satisfy the Monotonic Search heuristic, then there is an apparent lack of *forward progress* in the search for a proof. The Forward Progress heuristic is used to detect when a reasoning agent does significantly progress in its own local search efforts, and consequently, requests new information from other reasoning agents.

Once DARES has used the Forward Progress heuristic to determine that it should import some new information from other reasoning agents, it must then decide what information would be most beneficial. To accomplish this, DARES first applies a *Likelihood* heuristic to the clauses of the current resolution level. Those clauses which do not have the negated theorem in their ancestry are considered the least likely clauses to advance the proof and are assigned a likelihood of 0. The remaining clauses are assigned a likelihood equal to the reciprocal of the number of literals they contain. In this way, a single literal clause is assigned the maximum likelihood of 1 since obtaining the negation of this literal from another agent would immediately solve the proof. When making a request to import new knowledge from other reasoning agents, a DARES agent actually initiates a *cycle* of requests. The first request made to the other reasoning agents is for knowledge which will resolve with the clauses assigned the highest likelihood. If this request fails, the reasoning agent will repeatedly make similar requests for knowledge based on clauses which were assigned successively lower likelihoods. The form of each information request is a set of literals called the *Minimum Priority Negated Literal Set.* This set is obtained from a group of clauses with a particular likelihood, by first combining all the literals from each clause into a set. The literals which are instances of other literals are subsumed to minimize the amount

of information in the request. Finally, each literal in the set is finally negated for the convenience of the other reasoning agents.

When a reasoning agent receives a request for knowledge from another agent, it consults its database to find any clause which contains a literal that will unify with at least one literal in the Minimum Priority Negated Literal Set. If there is at least one clause found in the database, the reasoning agent sends a list of those clauses back to the requesting agent. When received by the requesting agent, this information is then tagged with the other agent's name and becomes incorporated in the reasoning agent's own local proof. If the requesting agent does not receive any new information from the other reasoning agents, even after the literals with minimum likelihood have been requested, the reasoning agent terminates its local search for a proof.

The Subsumption strategy plays a key role in controlling the size of the search space generated by an automated reasoning system. Subsumption is used to reduce the number of clauses in a list of resolvents by removing those clauses which are instances of another more general clause from the list. The procedure for performing subsumption on two clauses can be summarized as follows: a clause C1 is said to *subsume* another clause C2, if the variables of C1 can be instantiated such that *all* the resulting literals occur in C2[2]. As an example, consider applying the subsumption strategy to a list of two clauses; C1 = A(x), and C2 = (A(b) $\vee$ B(z)). Since clause C2 contains the literal B(z) and there is not a similar literal in clause C1, it is not possible for clause C2 to subsume clause C1. By assigning the variable x of clause C1 to the constant b, the only resulting literal A(b) is present in clause C2, therefore clause C1 subsumes clause C2. It is important to note that no information will be lost by removing clause C2 from the list. To see why this is so, consider the meaning of clause C1 which states that no matter what the value assigned to variable x is, the predicate A(x) is *always* TRUE. Consequently, any instance of clause C1, such as the literal A(b) which occurs in clause C2, must also be TRUE. Therefore, clause C2 is TRUE regardless of the value of B(z) and thus is reduced to an instance of clause C1.

3.2.2 Developing a Distributed Axiomatization

In the previous section, a brief description of how DARES reasoning agents prepared for a new reasoning task was given. While the precise nature of how reasoning agents are actually prepared was not made explicit, the level of detail given was sufficient to gain an understanding of the DARES system as a whole. In this and later sections, it would be helpful to become familiar with some of the more important details. As they are currently implemented, DARES reasoning agents are pre-programmed to accept various commands from external sources. Principally, these commands are given by other kinds of distributed agents. For a typical application, a DARES system would be used in conjunction with

[2] This description of subsumption was adapted from Exercise 2 on page 90 of [91].

other kinds of distributed problem solving systems as a mechanism for facilitating inter-agent reasoning. Generally, each DARES agent is considered to be uniquely paired with a specific agent from one of the other distributed systems. These agents are generally in control and manage their respective reasoning agents as a subordinate system. The basic commands understood by each reasoning agent include the following; 1) create and initialize a new theorem prover process, 2) load a theorem prover process with axioms and negated theorem, and 3) start a theorem prover process. Once a group of controlling agents has recognized the need for a distributed reasoning task, the members of the group must mutually agree upon a symbolic designator to uniquely identify the new task. From now on, when issuing commands to their respective reasoning agents, this symbol is referred to by each controlling agent in the group as the *theorem tag.* When a controlling agent has completed an axiomatization of its portion of a reasoning task, it then issues the above three commands in order, with each command being associated with the agreed upon theorem tag.

The axiomatization developed by each controlling agent affiliated with a particular distributed reasoning task will henceforth be referred to as a *local axiomatization.* Recall from section 3.1.1 a basic assumption of distributed problem solving which presumes no single agent has sufficient knowledge to completely solve a distributed problem without the co-operation of other agents. As a consequence, it can further be assumed the distribution of knowledge among any set of controlling agents is such that an axiomatization formulated by each agent must necessarily be incomplete. Thus, for a given distributed reasoning task, the axiomatization developed by each controlling agent is but a partial axiomatization and will reflect only that portion of the problem which is *locally* observable from an agent's perspective. A *distributed axiomatization* is considered to be the set of all local axiomatizations produced by a group of controlling agents for a single distributed reasoning task.

To describe in detail all the possible ways in which a distributed axiomatization may be devised to represent even a single distributed reasoning task would be exceedingly difficult. Unquestionably, a more pragmatic means of elucidating such a procedure would be to develop, in a step-by-step fashion, a distributed axiomatization for a typical distributed reasoning problem. Such a developmental process would not only serve to illustrate many of the difficulties typically associated with distributed reasoning problems, but would also show at least one way in which these difficulties may be overcome. A problem often used for such purposes is the classic *blocks world* problem. In the blocks world problem the object is to have an assembly robot arrange a stockpile of building blocks into a specific goal configuration. By adding more assembly robots, the blocks world problem can be made a suitable distributed problem for demonstrative purposes. In the distributed blocks world problem a number of independent robots must *coordinate* their interactions with one another in order to achieve a common goal configuration of blocks. In the following sections, a distributed axiomatization of the problem which automatically coordinates the actions of each assembly robot is developed.

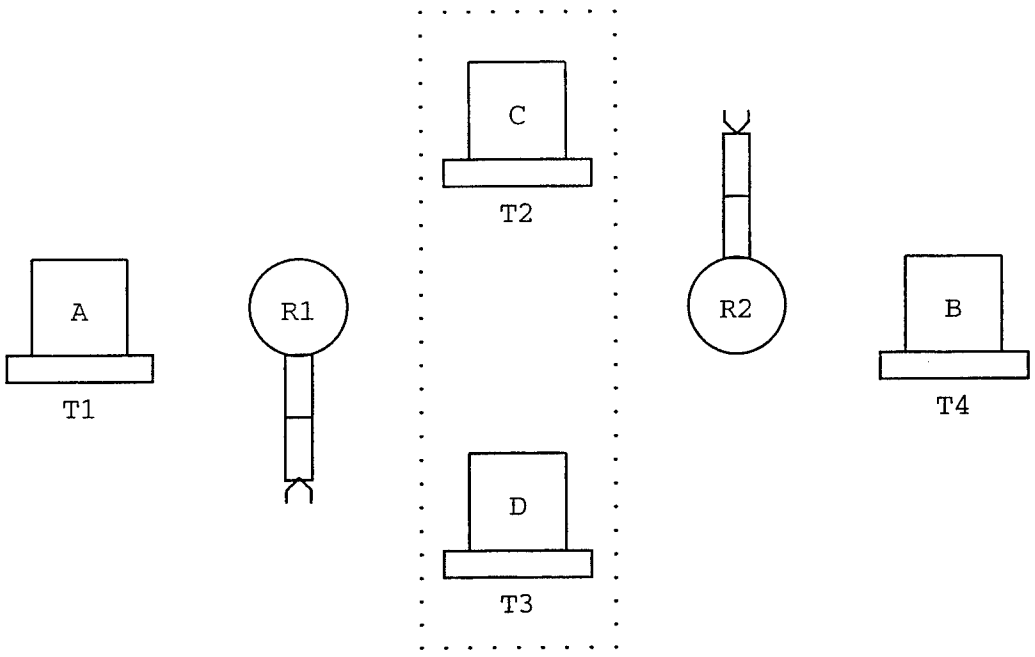


Figure 3.1: A Two-Robot Assembly Environment.

3.2.2.1 A Distributed Blocks World Problem

For the sake of simplicity, it will be assumed that all assembly robots are identical in form and function and that each robot manipulates blocks synchronously with respect to the others. Furthermore, it will be assumed that each assembly robot is (and always remains) stationary, much like many industrial robots. The *environment* of an assembly robot will be defined to include all physical objects of the distributed domain within its region of influence. Accordingly, the environments of any number of robots may overlap in an arbitrarily complex manner. As a matter of convenience in discussing the development of an exemplary distributed axiomatization, a particular instance of the distributed blocks world problem involving only two such assembly robots will be discussed and is illustrated in Figure 3.1. Throughout the following discussion, it is presumed the problem solving behavior of each physical assembly robot is being modeled by a corresponding distributed problem solving agent. It is these distributed assembly robot agents which actually produce the individual local axiomatizations that make up a particular distributed axiomatization. The local axiomatizations are then assumed to be given directly to a respective set of subordinate DARES reasoning agents in the manner described previously. Results from the distributed reasoning subsequently performed by the DARES system is what actually determines the actions of each assembly robot.

Figure 3.1 is a depiction of a two-robot blocks world problem as seen from a global

perspective. The objects to be manipulated are simple blocks labeled A, B, C, and D. The two robots labeled R1 and R2 are only capable of moving blocks directly from one table to another. Also, the tables labeled T1, T2, T3, and T4 are permanently attached to the floor of the robot assembly area and have only enough surface area to accommodate a single block being placed directly on them. Thus, the principle function a robot performs is one of picking up a block from one table and *stacking* it onto another table or block. For the purposes of developing an axiomatization, this sequence of robot actions will be simplified by consolidating them into a single stacking operation. When viewing the two-robot blocks world problem as a distributed problem, boundaries need to be established which delineate a region of responsibility for each individual problem solving agent. For the problem of Figure 3.1, the boundary between the regions of robot R1 and robot R2 can be visualized as a vertical line dividing tables T2 and T3 equally into two halves. Tables T2 and T3, as well as any blocks stacked on them, are what may be called *boundary* objects in that they are common to two or more (logically) adjacent regions of distribution. In Figure 3.1, the boundary objects common to the regions of both robots are shown enclosed by dotted lines.

The distribution of knowledge in the two-robot blocks world problem is such that properties of domain objects are maintained across distribution boundaries and remain consistent with the global problem. For example, an object which was a block named 'B' in the global problem is still a block named 'B' in every region of distribution it belongs to. As a consequence, an object labeled as 'C' in two different regions refers to exactly the same domain object. Likewise, the properties of 'C' will not conflict across distribution regions. For example, if one agent has knowledge that 'C' is an apple, no other agent will possess knowledge contrary to this. Each robot agent has complete knowledge of the domain objects within its designated region. Accordingly, both robot agents have knowledge of their own region's boundary objects. The definition of a robot's environment given previously parallels exactly the distribution boundaries for each robot agent established above. Therefore, the term "robot environment" will be used to designate a robot agent's region of responsibility. Robot R1's environment includes tables T1, T2, and T3, along with any blocks which may be on top of these tables. Similarly, the environment of robot R2 includes tables T2, T3, and T4, along with all the blocks on top of these tables. Herein lies most of the problem's difficulty, the overlap in robot environments translates to the boundary objects being simultaneously accessible to both assembly robots. Regulating their access to these domain objects constitutes a large portion of the distributed blocks world coordination problem.

Most distributed reasoning tasks fall into two broad general categories; 1) those which answer questions such as *"Yes the condition exists"* or *"No the condition does not exist"*, and 2) those which find complete solutions to a given problem. With regards to the type of reasoning required, the distributed blocks world problem, as it has been described, falls into the second category. The desired result of a reasoning task is a valid plan which, if faithfully followed by each assembly robot, will automatically *coordinate* their actions in achieving the desired assembly goal. In terms of defining an appropriate axiomatization for this kind

of reasoning response, the complications involved in representing a *state space search* must necessarily be introduced. When applying the method of state space search to distributed reasoning problems, a fundamental difficulty arises in maintaining a coherence between the local states of each reasoning agent and the global state of the distributed problem.

3.2.2.2 The Basics of a Local Axiomatization

While for demonstrative purposes the distributed blocks world problem under consideration has only two assembly robots, the axioms devised will not be limited to this special case as it would detract from the example's overall effectiveness. Wherever appropriate, the axioms will be structured in such a way as to accommodate a generalized n-robot problem. Turning now to the actual design of a distributed axiomatization, several types of predicates can immediately be identified which will be used to represent the kinds of physical domain objects associated with the distributed blocks world problem. In particular, these predicates are BLOCK(x) and TABLE(x). Their respective interpretations are "x is a BLOCK" and "x is a TABLE." A predicate ROBOT(x) could be defined as well, but for the axiomatizations which will be developed, it is not actually necessary. In order to facilitate a generalization of the developed axioms while at the same time keeping the number of required predicates to a minimum, it will be convenient to identify each robot using a recursive function representation. The *successor function* can be used for this purpose and will later be exploited to impose an ordering on the selection of simultaneous robot actions. In number theory, the successor function $s(x)$ is defined as $s(x) = x + 1$ for all x and can be used as a means of counting. More importantly for the purposes of naming individual robots, the successor function can be used to conveniently represent integers recursively in the following way: $s(0) = 1$, $s(s(0)) = 2$, $s(s(s(0))) = 3$, and so on, where the level of functional recursion is equal to the desired integer. When assigning labels to the various assembly robots, their respective robot agents will substitute a corresponding recursive successor function representation. For example, robots which would normally be labeled R1, R2, and R3 will be named respectively $s(0)$, $s(s(0))$, and $s(s(s(0)))$ in a distributed axiomatization. The beginnings of a distributed axiomatization for the two-robot assembly problem of Figure 3.1 can now be tabulated. In the following, axioms listed on the left form a partial local axiomatization for robot R1 while those on the right form a partial local axiomatization for robot R2.[3]

[3] Throughout this section , axioms which are part of a distributed axiomatization will be labeled consecutively with positive integers prefixed with an 'A'. Likewise, clauses will be labeled using positive integers prefixed with a 'C'. Additionally, some axioms and clauses will also be marked with a prime (') or double prime (") to distinguish those which are specific to either of robot R1 or R2 respectively.

Robot R1		Robot R2	
BLOCK(a)	(A1′)	BLOCK(c)	(A1″)
BLOCK(c)	(A2′)	BLOCK(d)	(A2″)
BLOCK(d)	(A3′)	BLOCK(b)	(A3″)
TABLE(t1)	(A4′)	TABLE(t2)	(A4″)
TABLE(t2)	(A5′)	TABLE(t3)	(A5″)
TABLE(t3)	(A6′)	TABLE(t4)	(A6″)

A rudimentary set of predicates for the distributed blocks world problem should be sufficiently expressive to adequately describe all primitive aspects of the domain. One aspect of the blocks world problem which has not yet been expressed in predicate form is the existence of various relationships among domain objects in which one object is on top of another. Many axiomatizations of the classic blocks world problem have used a pair of predicates such as ON(x y s) and CLEAR(x s) to describe these relationships, and these will be used in this distributed axiomatization as well. Respectively, their interpretations are "x is ON y in *state* s" and "x is CLEAR in *state* s." Both predicates are dependent on the current state of the problem domain and will become the cornerstones in searching the problem's state space for a solution. Conceptually, when the assembly robots synchronously move their respective blocks, they are transforming the previous state of the world into the next current state. In this new state, blocks which were on top of other objects, or objects which were clear, may no longer be so. Therefore, the only valid ON and CLEAR properties are those which are true in the current state of the world. For each robot, the initial state will be designated by the constant symbol 'end'. The choice of 'end' is entirely arbitrary and is intended to convey the meaning of being the end of a list. Successive states will be represented as lists which are built upon previous state representations. Lists are constructed using a functional representation similar to that of the successor function discussed previously. In this respect, the initial state 'end' will serve as an "anchor" for subsequent state representations in much the same way that 0 "anchored" the successor function. The relationships among the domain objects in the current state can now be expressed for each robot by the following additional axioms:

Robot R1		Robot R2	
CLEAR(a end)	(A7′)	CLEAR(c end)	(A7″)
CLEAR(c end)	(A8′)	CLEAR(d end)	(A8″)
CLEAR(d end)	(A9′)	CLEAR(b end)	(A9″)
ON(a t1 end)	(A10′)	ON(c t2 end)	(A10″)
ON(c t2 end)	(A11′)	ON(d t3 end)	(A11″)
ON(d t3 end)	(A12′)	ON(b t4 end)	(A12″)

Due to the distributed nature of the blocks world problem under consideration, yet another predicate needs to be introduced before the state of the problem domain can be

completely described. In anticipation of the need for such a predicate, the environment of a robot was previously defined and now becomes a convenient way of representing the distribution of domain knowledge in the current state. Conceptually speaking, when robot A removes an object from the environment of robot B, robot A has in effect redistributed the domain knowledge since robot B does not possess knowledge of the object in the next state. The predicate ENVIRONMENT(x y s) which has the interpretation "object x is in the ENVIRONMENT of robot y in state s" will be used for this purpose. With the addition of this predicate there is now sufficient representation to completely describe the problem domain in any state. Adding the following axioms to the local axiomatizations of the respective robots will complete the descriptions of their initial states.

Robot R1:

$$(\forall z)ENVIRONMENT(t1\ s(0)\ z) \tag{A13'}$$
$$(\forall z)ENVIRONMENT(t2\ s(0)\ z) \tag{A14'}$$
$$(\forall z)ENVIRONMENT(t3\ s(0)\ z) \tag{A15'}$$

Robot R2:

$$(\forall z)ENVIRONMENT(t2\ s(s(0))\ z) \tag{A13''}$$
$$(\forall z)ENVIRONMENT(t3\ s(s(0))\ z) \tag{A14''}$$
$$(\forall z)ENVIRONMENT(t4\ s(s(0))\ z) \tag{A15''}$$

A complementary relationship exists between the ON and CLEAR predicates in that an object which is clear does not have any other object on it, and an object which has another object on it is not clear. This relationship can be expressed concisely with a *well-formed formula* (wff) from the predicate calculus and will become the first nontrivial axiom in the local axiomatizations of each assembly robot. The axiom can be written as follows:

$$(\forall xz)(CLEAR(x\ z) \Leftrightarrow \neg(\exists y)ON(y\ x\ z)). \tag{A16}$$

For an automated reasoning program however, the handling of such expressions can be cumbersome as well as time consuming. Instead, most automated reasoning systems require logical statements to be written in a language closely related to that of the predicate calculus called the language of clauses. Fortunately, there exists a relatively straightforward procedure for transforming wffs of the predicate calculus into corresponding sets of *clauses.* Essentially, this transformation procedure involves manipulating a wff into an equivalent *prenex normal form*[4] in which all quantifiers have been brought to the "outside". The resulting formula which now lies within the scope of every quantifier is subsequently transformed

[4] A wff $(Q_1y_1)\ldots(Q_ny_n)\mathcal{A}$, where each $(Q_iy_i$ is a universal or existential quantifier, y_i is different from y_j for $i \neq j$, and $\mathcal{A}$ contains no quantifiers, is said to be in *prenex normal form.* (Includes the case $n = 0$ when there are no quantifiers.) [60, page 82]

into an equivalent *conjunctive normal form*— a conjunction of disjunctions.[5]Existential quantifiers are then removed and each disjunction is taken to be a single clause. In practice however, it is not necessary for a wff to be put into prenex normal form. Simply renaming the variables of each quantifier so that each variable is given a different name will serve the same purpose. Each existential quantifier present in the formula can then be removed by consistently replacing all occurrences of the quantified variable with a suitable *skolem function.* A skolem function is used to name whatever it is which exists. For example, in the wff $(\forall x)(\exists y)P(x\ \ y)$ the existentially quantified variable y can be replaced by a function $f(x)$ as in $(\forall x)P(x\ \ f(x))$. The value of $f(x)$ is considered to be the name of an object y which exists and will make $P(x\ \ y)$ true. It is necessary to replace each existentially quantified variable with a uniquely named skolem function. In addition, skolem functions must have as many parameters as there are universally quantified variables which enclose the existential quantifier as was done in the above example. The reason is the value of an existentially quantified variable may depend on the values of any enclosing universally quantified variables.

Clause representations of axioms and theorems are more easily manipulated internally by an automated reasoning program in that they lend themselves to a purely mechanical and uniform processing.

The above axiom can be expressed using clauses as follows:

$$\neg CLEAR(x\ z) \ \mid\ \neg ON(y\ x\ z) \qquad \text{(C16)}$$

$$ON(y\ x\ z) \ \mid\ CLEAR(x\ z) \qquad \text{(C17)}$$

Notice, for example, how by resolving clause C7′ (same as axiom A7′) with the above would produce the clause $\neg ON(y\ a\ end)$ which can be interpreted as *"there is no object on top of object 'a' in the 'end' (initial) state."* Strictly speaking, the CLEAR predicate is not really necessary since it merely translates to a negated ON predicate. When devising axioms and theorems, expressing such relationships in a positive sense is often desirable since this is less confusing and often eliminates errors.

In the representations of each assembly robot's initial state of the world as given above, it may appear as though a number of axioms describing the extent of their environments has been omitted. Specifically, axioms which explicitly assert that certain blocks are in the environment of a particular robot are absent from these representations. As a result of the assumptions that all robots and tables in the assembly area are fixed in their location, a universal axiom such as Axiom A13′ can be used as a basis for inductive reasoning. Given representations of world states like those for robots R1 and R2 above, there exists sufficient information to *automatically* derive all knowledge of which robot environments a particular

[5] A form is in *conjunctive normal form* (cnf) if it is a conjunction of one or more conjuncts, each of which is a disjunction of one or more literals—for example, $(B \vee \neg C) \wedge (A \vee D)$, A, $A \wedge B$, $A \vee \neg B$, and $A \wedge (B \vee A) \wedge (\neg B \vee A)$. [60, page 25]

block is a member of. Using Axiom A13′ as a base case for an induction, it can be inferred that any object which is on another object, known to be in a particular robot environment, must also be in that same environment. In the local axiomatization produced by each assembly robot, this induction of knowledge is provided by the following axiom:

$$(\forall uvxz)(ON(u\ v\ z) \wedge ENVIRONMENT(v\ x\ z) \Rightarrow \\ ENVIRONMENT(u\ x\ z)) \qquad \text{(A17)}$$

Employing a robust axiom such as this will prove to be quite advantageous as it will allow the state transformation axioms, which will be discussed later, to be significantly simplified.

3.2.2.3 Controlling Robot Agent Interactions

As mentioned earlier, some kind of scheme is needed for controlling each robot's access to boundary objects in its current environment. Simply writing an axiom which imposes mutually exclusive access to these objects would not be effective. An analogy can be drawn between such an axiom and a multiprocessing computer program which requires access to a shared variable. The normal procedure for gaining access to the variable would be to first check its status flag to determine whether or not access is allowed. If access is allowed, a processor must then change the variable's status flag to block other processors from accessing the variable. Without some means of localizing controlled access to the variable, there can be no guarantee that between the time when the processor checked the status flag and subsequently changes it, that another processor has not already done so. Typically, this problem is solved by extending the memory bus cycle to include both a read and subsequent write operation of the status flag's memory location. Likewise, it is possible for a number of robot agents to each believe they are the agent which has exclusive use of the same domain object. There are two possible ways in which this difficulty may be overcome: 1) allow each robot agent to choose the block it will move independently of the other agents and backtrack to resolve conflicts when they are detected, or 2) impose an ordering on the robot agents such that an agent selects a block to move only after all the robot agents before it have made their selections.

Setting aside the question of how to maintain globally coherent states, independently selecting blocks becomes very expensive, computationally, when these selections are found to be incompatible. Adopting such a strategy when using an automated reasoning system, with its combinatorially explosive nature, would be ill-advised. However, the latter approach is much more straightforward and is a reliable method of ensuring globally coherent search states. This methodology for controlling robot agent interactions is based on the assumption that an effective procedure exists for establishing a global ordering recognized by all involved agents. For the distributed blocks world problem under consideration, there are only two robots labeled R1 and R2 respectively. These labels were assigned *a priori* when the problem was first defined. Since both robots are expected to participate in the solution of the

problem, establishing an ordering of agents is a simple matter of recognizing that robot R1 selects its block before robot R2. So long as all assembly robots are involved in the problem and they are consecutively named, an *a priori* ordering can be extended to any number of assembly robots. Otherwise, it must be assumed the group of agents working on the problem are capable of developing such a consecutive ordering, or some equivalent.

Recall from an earlier discussion in which it was described how the robot agents substituted their respective robot names for a corresponding recursive successor function representation. Such a functional representation is very convenient for implementing a robot agent ordering *without* knowing *a priori* how many robots are participating in the problem. For reasons which will become clear shortly, the ordering of robot agents is such that those with names less than n must all select blocks to move before the robot agent named n is allowed to make its selection. With respect to the robot agent named n, all other robot agents less than n are predecessors with the robot agent named $n-1$ being the immediate predecessor. As will become evident in the following discussion, only the immediate predecessor relationships are strictly necessary to the formulation of a local axiomatization. By cleverly choosing a representation for robot names, knowledge of these immediate predecessor relationships have been purposefully encoded into the name of each robot. For example, any robot whose name can be represented as $s(x)$, implicitly carries the knowledge that its immediate predecessor's name is x.

3.2.2.4 Searching for Globally Coherent States

Using the ordering technique for controlling robot agent interactions suggested in the previous section, a methodology for searching distributed state spaces can now be developed. The basic idea is to implement a two-level search procedure—a kind of search within a search. At the root level, descendants of a given valid global state are produced by searching the space of all combinations of simultaneous robot activities. A globally coherent state is generated by incrementally incorporating actions for each robot to perform within a *single* synchronous time step. This incremental procedure progresses according to the assumed *a priori* ordering imposed on the robot agents. A better understanding of this portion of the search strategy can be gained by visualizing the chain of robot agent immediate predecessor relationships as a series of directed line segments spanning the network of involved agents as depicted in Figure 3.2. With respect to the figure, if it is assumed the agent labeled 'A' is the predecessor of all other agents, then agent 'A' would always select first from the set of possible robot actions in the current global state. After agent 'A' has made its selection, agent 'B' can then make its selection from a reduced set of possible robot actions. The set of possible robot actions is effectively reduced after each robot makes its selection in that each succeeding robot must avoid choosing an action which would conflict with one chosen by a predecessor robot. When the last robot agent, agent 'F', selects its action to perform, the set of individual actions chosen by each participating robot agent represents a new (and

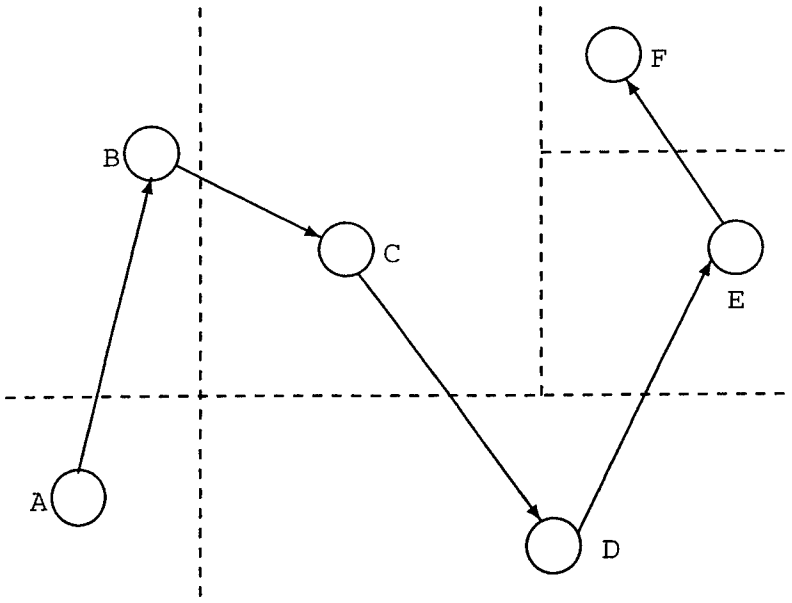


Figure 3.2: Chaining of Robot Agent Superiority.

globally coherent) state in the distributed problem's search space.

In order to express in a set of axioms the incremental development of globally coherent search states as described above, it is necessary to augment the existing repertoire of predicates. A number of new predicates need to be devised which would capture this notion of incrementally constructing a new global state. The first such predicate to be introduced is CONTEXT(x y s) which will be interpreted as meaning "robot x has chosen its action and produced CONTEXT y in the current global state s." A second predicate LEGAL(x y z s) will be understood to mean "it is legal for robot x to choose an action involving object y for the context z in the current global state s." Setting aside for the moment an axiomatic definition of LEGAL, the CONTEXT predicate may be defined as follows:

$$(\forall uvwxz)(CONTEXT(x\ f(a(u\ v)\ w)\ z) \Leftrightarrow (BLOCK(u) \wedge LEGAL(x\ u\ w\ z) \wedge LEGAL(x\ v\ w\ z) \wedge \neg EQUAL(u\ v))) \quad \text{(A18)}$$

The functional argument $f(a(u\ v)\ w)$ in the above definition is of a recursive form which simultaneously represents the newly generated context as well as the action chosen by robot x which generated the context. In this notation, the action performed by robot x, denoted by the function $a(u\ v)$, is interpreted as meaning object u is placed on top of object v. The arguments of function f will be understood to convey the knowledge "action $a(u\ v)$ was selected in context w." Using a function in this manner is a representational "trick" often practiced when writing axioms for an automated reasoning system. In effect, it implements a recursive list structure where the leading argument(s) is the data which constitutes the

list's first element. The last argument of the function is usually reserved for holding the remainder of the list in a manner similar to the CDR of a LISP list structure. The above axiom may be thought of as the top-level axiom responsible for generating all possible combinations of placing a block on another domain object. One possible action which a robot may choose to perform during any synchronous time step, is the "action" of remaining *idle*. This possibility can be accommodated with the following axiom:

$$(\forall xyz)(CONTEXT(x\ y\ z) \Rightarrow CONTEXT(s(x)\ f(idle\ y)\ z)). \tag{A19}$$

A definition of the LEGAL predicate must rely heavily on the detection of any proposed robot actions which would conflict with the actions of its predecessors. To represent such contention for the domain objects, a new predicate CONFLICT($x\ y\ z\ s$) will be defined. This new predicate has the interpretation "an action by robot x which involves object y is a CONFLICT in context z of the current global state s." Because it has been assumed the assembly robots are completely synchronous, it could be argued that while one robot has picked up and is presently moving a particular block, a second robot could place its block where the first block once was. Allowing such actions to occur however, would significantly complicate the axiomatizations of each robot, which is not the aim of this example. Therefore, with the exception of an idle action, involving either of the domain objects required by the actions of a predecessor robot will be considered a conflict. In any given context, determining which of the domain objects would cause a conflict, were they manipulated by a particular robot, is accomplished using two pieces of information readily available. The most immediate of which is found in the context generated by a robot's immediate predecessor. Involving either of the objects associated with an immediate predecessor's action would be a conflict. This is the main mechanism for deriving new conflict relationships. The remaining conflicts are generated through inheritance from the immediate predecessor's set of conflict relationships. Certainly, manipulating an object which was a conflict for a robot's immediate predecessor, is also a conflict for the robot. An axiomatic definition of CONFLICT is given below;

$$\begin{aligned}(\forall uvwxyz)(&CONFLICT(s(x)\ y\ f(a(u\ v)\ w)\ z) \Leftrightarrow \\ &(CONTEXT(x\ f(a(u\ v)\ w)\ z) \wedge (EQUAL(y\ u) \vee \\ &EQUAL(y\ v) \vee CONFLICT(x\ y\ w\ z)))).\end{aligned} \tag{A20}$$

The definition of LEGAL can now be given in terms of the existing inventory of predicates. For a robot to legally involve an object in a proposed action, it must first be accessible to the robot. In the current global state, the object should at least be in the environment of the particular robot as well as be clear. Additionally, in the context generated by the particular robot's immediate predecessor, using the object should not be a conflict. These properties are used to define what it means to be legal in the following axiom:

$$(\forall wxyz)(LEGAL(s(x)\ y\ w\ z) \Leftrightarrow$$

$$(ENVIRONMENT(y\ s(x)\ z) \wedge CLEAR(y\ z) \wedge$$
$$CONTEXT(x\ w\ z) \wedge \neg CONFLICT(s(x)\ y\ w\ z))) \quad \text{(A21)}$$

In the preceding axiomatic definitions that were developed for predicates CONTEXT, CONFLICT, and LEGAL, an inductive assumption has been made; that is, given a context created by some robot and a corresponding set of conflicts, a new context for the succeeding robot can be generated. If this is not perfectly clear, it may help to refer back to Figure 3.2. In the figure, the inductive assumption may be visualized in the following way; suppose a partial global state has somehow been developed from agent A along the segmented line to agent C. Using these predicates, this partial solution may be further developed by extending it to include agent D. Once begun, such an inductive procedure could be carried out to include any number of participating robot agents. To complete the procedure for developing globally coherent search states, two additional requirements need to be satisfied; 1) a means of initiating the search procedure must be devised, and 2) a method for recognizing valid global states must be established. In order for the first robot to initiate the procedure by generating a new context, suitable conditions must be supplied which would "enable" a set of LEGAL predicates to be derived. Since no other robot has previously selected an action, it is of course legal for the selected action of the first robot to involve any domain object within its environment. Referring back to Axiom A21, it can be seen that an initial context and conflict axiom are all that is required. The following two axioms will suffice:

$$(\forall z)CONTEXT(0\ end\ z) \quad \text{(A22)}$$
$$(\forall xz)\neg CONFLICT(s(0)\ x\ end\ z). \quad \text{(A23)}$$

Axiom A22 represents an initial context which will initiate the search for new global states given *any* current global state. Note this context was generated by some *fictitious* robot named 0 (i.e. the immediate predecessor of the first robot R1 assigned the name $s(0)$). As with the initial global state, all initial contexts will be represented by the constant 'end'. Axiom A23 is used to establish the fact there are never any conflicts for the first robot when it selects an action.

A newly generated globally coherent state can be recognized by examining each new context for the name of the robot which generated it. If the robot name corresponds to the name of the last robot in the assumed *a priori* ordering, then the particular succession of context generation has produced a new valid global state. Any such context will be referred to as a *terminal* context. The information contained in a terminal context can be thought of as a kind of program for issuing directives to each participating assembly robot. This program essentially informs each respective robot that it is now permissible to create a new search state by performing the action it selected during the construction of the terminal context (i.e. the new global state). A new predicate STATE($x\ y\ z$) will be introduced to facilitate the execution of such a program. Its interpretation is "the robot which generated context x can create a new search state by performing its corresponding

selected action in the current state z." The argument y is used to distribute an intact version of the terminating context to each participating robot agent so that it may use this context to generate a label $g(y\ z)$ for the new global state. The following axiom is used to recognize each new terminal context:

$$(\forall wxz)(MAXROBOT(x) \wedge CONTEXT(x\ w\ z) \Leftrightarrow STATE(w\ w\ z)). \tag{A24}$$

The predicate MAXROBOT(x) is used to identify which of the participating assembly robots has been assigned the name of the last robot in the assumed *a priori* ordering. While each robot agent can refer to this predicate, as was done above, only the robot agent understood to be last in the ordering will actually have an axiom which instantiates the predicate with its assigned name. Notice that Axiom A24 does not actually terminate a succession of robot context generation, it merely "marks" a particular robot context as representing a terminal point in the sequence of real physical robots.

A second axiom is needed to "breakdown" the information of a terminal context so that the appropriate assembly action directives may be recognized by each predecessor of MAXROBOT(x). With respect to Figure 3.2, this process may be conceptualized as reversing the direction of travel along the segmented line connecting each participating assembly robot agent. As the terminal context is successively broken down, a new STATE predicate needs to be generated for each predecessor robot along the way. Programming of the assembly robots with a terminal context will be complete when a STATE predicate is generated for the first robot in the assumed ordering. The following axiom can be used to accomplish the task of breaking down a terminal context and programming each predecessor robot:

$$(\forall vwxz)(STATE(f(v\ w)\ x\ z) \Rightarrow STATE(w\ x\ z)). \tag{A25}$$

3.2.2.5 Implementing State Transformations

Were the Blocks World problem under consideration not distributed, a simple search of the problem's state space would have yielded a solution to the problem providing one did exist. However, in the distributed problem, the primary obstacle in implementing a state space search is discerning an appropriate distributed state space to search. This obstruction was overcome in the previous section by effectively circumscribing just such a distributed state space. An inductive procedure was developed which produces all possible simultaneous robot actions permissible in any state of the distributed problem. Referring back to Section 3.2.2.2, the axioms dependent on the current global state, by default uniformly designated the initial state of the distributed Blocks World problem as the symbolic constant 'end'. Similarly, the local axiomatizations produced by any number of participating robot agents would also designate their initial global states by the constant 'end'. By doing so, any number of robot agents can establish an initial (or "root") global state which they can

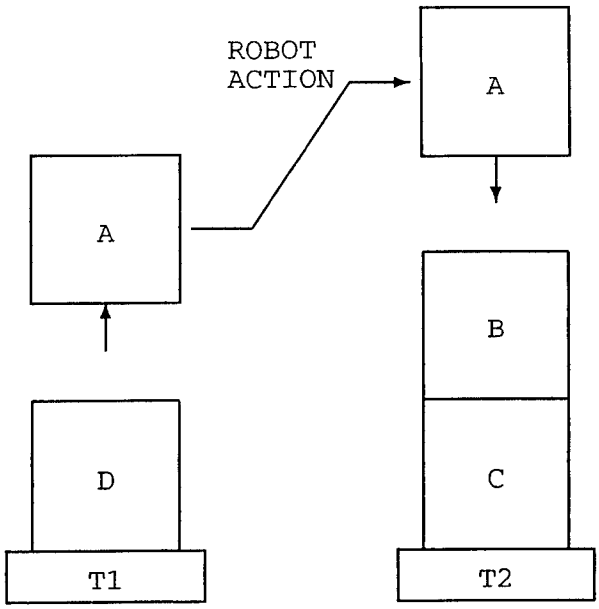


Figure 3.3: A Hypothetical Robot Action.

all refer to by the same name. From this initial global state, branching of the search tree is accomplished by applying *global* state transformations corresponding to the valid simultaneous actions of a terminal context. In the previous section, the STATE predicate was introduced as a mechanism for breaking down global state transformations into their constituent transformations of individual robot environments. Each STATE predicate imparts knowledge sufficient for a robot agent to complete a *local* transformation of its environment and effect an overall global state transformation.

A number of axioms will need to be developed in order to bring about the necessary robot environment transformations. To gain some insights into the details which must be accommodated when performing a transformation, consider the hypothetical situation depicted in Figure 3.3. In this figure, the robot agent has been directed to move block A from on top of block D on table T1 over to the top of block B on table T2. The table below is a partial summary of the robot's environmental state before and after the action is performed.

Before		After
CLEAR(a s)	$\Rightarrow$	CLEAR(a s')
ON(d t1 s)	$\Rightarrow$	ON(d t1 s')
ON(b c s)	$\Rightarrow$	ON(b c s')
ON(c t2 s)	$\Rightarrow$	ON(c t2 s')
CLEAR(b s)		CLEAR(d s')
ON(a d s)		ON(a b s')

Notice how the transformation of the robot's environment preserved the first four properties from the previous state s by including them in the new state s'. The retained ON properties can be seen to correspond with descriptions of the stacked blocks *beneath* the block which was moved (i.e. block A) before *and* after the action took place. This domain characteristic can be exploited so that axioms may be devised which will readily preserve such relationships. In addition to the properties retained during an environment transformation, several new properties are also introduced which directly reflect the particular action being performed. Specifically, block D is now CLEAR and block A is now ON block B. The following axiom initiates an environment transformation by asserting the existence of properties which are a direct consequence of a particular action.

$$(\forall uvwyz)(STATE(f(a(u\ v)\ w)\ y\ z) \Rightarrow (CLEAR(u\ g(y\ z)) \wedge ON(u\ v\ g(y\ z)) \wedge (\forall x)(ON(u\ x\ z) \Rightarrow CLEAR(x\ g(y\ z))))) \quad \text{(A26)}$$

Including the assertion which states the moved block remains CLEAR in the new state in Axiom A26, was purely a matter of convenience. With respect to the hypothetical situation depicted in Figure 3.3, the new environment properties of block D is CLEAR and block A is ON block B can be used to activate an induction for deriving the persistent properties of the environment transformation. Consider the stack of blocks on table T2 which block A has been placed on top of. In keeping with assumptions made previously which placed limitations on the allowable modes of robot interaction, there is no need to consider situations in which several robots may have manipulated blocks of the same stack in any synchronous time step. Therefore, a robot agent can readily use the new environment property $ON(a\ b\ s')$ asserted by Axiom A26 in conjunction with the old property $ON(b\ c\ s)$ to infer that block B *must still* be ON block C in the new state. Similarly, this new property can then be used to infer others until all the ON properties corresponding to the stack of blocks under the moved block have been carried over to the new state. An axiom which provides such an induction is the following:

$$(\forall uvwyz)(ON(u\ v\ g(y\ z)) \wedge ON(v\ w\ z) \Rightarrow ON(v\ w\ g(y\ z))) \quad \text{(A27)}$$

Axiom A27 provides a natural means of carrying out the desired state transition for the

stack of blocks which are under the moved block in the new state. This induction procedure can be extended to include the stack of blocks which were under the moved block in the previous state as well. Referring once again to the hypothetical situation of Figure 3.3, two pieces of information concerning block D are readily available. The first piece of available information was provided by Axiom A26 which states that block D is CLEAR in the new state, and the second piece of information available is that block A was ON block D in the previous state. Considering the restrictions placed on robot interactions, it can be assumed that whatever domain object block D was on in the previous state, it must also be on in the new state. The following axiom is used to generalize a recognition of this relationship;

$$(\forall uvyz)(ON(u\ v\ z) \wedge CLEAR(v\ g(y\ z)) \Rightarrow \\ (\forall w)(ON(v\ w\ z) \Rightarrow ON(v\ w\ g(y\ z)))) \qquad \text{(A28)}$$

Axiom A28 will assert, at most, a single ON relationship for a CLEAR block in the new state. This is all that is necessary since an ON property in the new state becomes a basis for Axiom A27 to inductively assert the remaining properties which are carried over into the new state. With the addition of Axiom A28, the local axiomatization provided by each robot agent is sufficiently complete to effect a local state transformation involving *only* those domain objects associated with a particular robot action. However, since a robot may choose to be idle during any synchronous time step, the possibility always exists that a stack of blocks on some table (or the table itself) will not be involved in any robot's action. The axioms developed to this point are not capable of coping with this situation and therefore may not always bring about a global state transformation. Fortunately, there is information available which can be used to detect such a condition should it ever occur. The key is Axiom A20 which defines exactly when involving an object in a robot action would conflict with that of a superior robot. Stacks of objects left untouched after a synchronous time step can be identified by considering which of the domain objects could be involved in an action by a *successor* of MAXROBOT(x), if one existed. The context parameters carried by CONFLICT predicates applicable to the direct superior of MAXROBOT(x), correspond to a terminal context in the particular global state. In any global state, all domain objects which are CLEAR and may be manipulated by some *fictitious* direct superior of MAXROBOT(x), must also be CLEAR in the next corresponding global state. This knowledge is sufficient to identify which blocks were left untouched on top of a stack as well as any untouched empty tables. A similar condition to Axiom A28 can be used to transform the ON properties of blocks beneath a CLEAR untouched block and can be stated as in the following axiom;

$$(\forall uvxyz)(MAXROBOT(x) \wedge \neg CONFLICT(s(x)\ u\ y\ z) \wedge CLEAR(u\ z) \Rightarrow \\ (CLEAR(u\ g(y\ z)) \wedge (\forall v)(ON(u\ v\ z) \Rightarrow ON(u\ v\ g(y\ z))))) \qquad \text{(A29)}$$

Axiom A29 completes the set of axioms needed to bring about comprehensive global state transformations. It may appear as though some axioms are needed to update which

robot environments each domain object belongs to in the new state. However, as was mentioned previously in Section 3.2.2.2, Axiom A17 is capable of inductively asserting the necessary relationships using the ON properties established for each new state.

3.2.3 Instituting a Distributed Solution Criteria

The only remaining aspect of each local axiomatization which must be addressed, is that of establishing a global goal condition. Simply stated, a global goal condition is a single theorem which is supplied to each DARES reasoning agent that represents the overall goal of a distributed problem. For example, in the case of the distributed Blocks World problem, a possible theorem (i.e. reasoning objective) may be stated as "it is possible to achieve a distributed problem state in which block A is on block B, and block B is on table T1." A proof of this theorem would determine one possible set of actions each assembly robot could perform which would transform the initial problem state into one satisfying the theorem. If a theorem represents a global goal and is known throughout the distributed domain, each robot agent could complete its local axiomatization simply by including the negated form of the theorem. The theorem must be negated to facilitate the refutation method of proof used by DARES. While it may be feasible in certain distributed domains to provide global knowledge of the desired goal condition, suitable axioms can be included in each local axiomatization which would achieve the same effect for the general case in which the goal condition is also distributed. With reference to the distributed Blocks World problem under consideration, robot R2 may have partial knowledge of the goal condition reflecting its view of the distributed problem which states that block B should be ON table T2. Likewise, R1 may have partial knowledge of the goal condition which states that block A should be ON block B and CLEAR. Any valid problem state which simultaneously satisfies both partial goal conditions, can be said to satisfy the global goal condition and therefore represents a solution.

A new predicate PARTIAL(z x) can be used by each robot agent to essentially "mark" each global search state which meets the requirements of its partial goal condition. It has the interpretation "the global search state z satisfies the partial goal condition of robot x", and can be defined as;

$$(\forall z)(PARTIAL(z\ x) \Leftrightarrow \mathcal{A}(x)). \tag{A30}$$

The term $\mathcal{A}(x)$ is used to denote a problem dependent wf which is used by each robot agent to describe its partial knowledge of the global goal condition. For instance, in the example given above, the agent corresponding to robot R1 would produce the following axiom:

$$(\forall z)(PARTIAL(z\ s(0)) \Leftrightarrow ON(a\ b\ z) \wedge CLEAR(a\ z)$$

Using a method similar to the one developed in Section 3.2.2.4 for incrementally deriving valid global states, a second predicate SOLUTION(z x) can be used to represent an

incremental conjunction of the partially satisfied goal conditions. This new predicate has the interpretation "the global search state z satisfies the partial goal conditions of all robots upto and including x." An axiom suitable for incrementally extending a partial solution is defined as follows:

$$(\forall xz)(SOLUTION(z\ x) \wedge PARTIAL(z\ s(x)) \Leftrightarrow SOLUTION(z\ s(x))) \qquad \text{(A31)}$$

As with the other inductive axioms presented thus far, an initial condition is required to begin the process and is given as follows:

$$(\forall z)SOLUTION(z\ 0) \qquad \text{(A32)}$$

The only remaining axiom needed to complete the local axiomatization of each robot, and hence a distributed axiomatization, is a specification of the global goal condition. With the above axioms, a global theorem which may be asserted by each robot agent can be written as follows:

$$(\exists xz)(MAXROBOT(x) \wedge SOLUTION(z\ x))$$

Negating this theorem for DARES produces the following:

$$(\forall xz)\neg(MAXROBOT(x) \wedge SOLUTION(z\ x)) \qquad \text{(A33)}$$

3.3 An Expert System Approach

Throughout the development of the distributed axiomatization presented in the previous section, a body of knowledge was drawn upon which exists above and beyond any fundamental knowledge of the problem domain. This body of knowledge does not represent knowledge of objects in a problem domain and their properties, but rather it represents knowledge of the problem domain itself—a kind of meta-level knowledge. For example, two pieces of meta- level knowledge used to develop the axiomatization of the distributed Blocks World problem were: 1) the names of each robot were purposefully assigned to correspond with an *a priori* ordering of the distributed agents, and 2) the agent corresponding to the last robot in the ordering has knowledge to that effect. Such knowledge is typically accessible to *all* distributed agents and can be viewed as a unifying *thread* by which the problem solving agents are intimately connected. In other words, a body of readily accessible meta-level knowledge can be used by a homogeneous distributed problem solving system as a basis for a uniform and consistent treatment of certain aspects of the problem domain. A case in point is the use of the two pieces of meta-level knowledge described above. The local axiomatization produced by each robot agent effectively applied this knowledge in a uniform and consistent manner to distributively implement a search for globally coherent distributed Blocks World problem states. In the next section we show how these ideas can be implemented in a system of distributed expert agents.

3.4 System Implementation

3.4.1 The Simulation Environment

SIMULACT [58] is a domain independent distributed simulation environment for developing and experimenting with distributed problem-solving systems. The active elements of a simulation are thought of as being like a group of characters performing on stage. There are two basic types of characters, *actors* and *ghosts*, which are used to model the characteristics associated with a particular distributed problem-solving domain. A simulation is controlled in much the same way as a theatrical drama is produced for the stage. The actions of individual actors and ghosts are supervised by a *director* and proceed in accordance with a pre-defined *script function.* A character's script function determines its problem solving role and defines how it may interact with other characters. These interactions are in the form of discrete messages communicated from one character to another and are expedited in SIMULACT by a mechanism analogous to a stage manager. The director is a control structure which integrates the activities of each character into an appropriate sequence of simulated actions.

Actors are designed to model a single problem-solving agent in a network of distributed processing nodes. Each actor is implemented as a distinct and independent process which executes a script function especially tailored for the requirements of the agent being modeled. Agents of distributed networks are generally thought of as being persistent. That is, they remain present throughout any and all problem- solving activities. When this is the case, the script function for an agent is typically structured as an infinite loop which dispatches and receives various control messages. The response of an agent evoked by control messages is based on the problem-solving nature of the particular agent and the abilities it has been endowed with. For a group of homogeneous agents, SIMULACT provides a *support package* facility which enables a group of agents to share a common framework or set of capabilities. As required by its task, an actor may inherit functional components from any number of support packages. The complete specification of all the actors and ghosts as well as individual support packages is described by a *world* file. SIMULACT uses such files to create and initialize simulations.

Many real-world distributed problems are complicated by the presence of extraneous information, and/or may exhibit some anomalous behavior which in order to be properly accounted for, must be injected into a simulation. It is often the case that these kinds of *events* represent an instance of a distributed problem and thus become the catalyst for some problem-solving activities. Such events may readily be compiled into a temporal sequence, called an *event scenario*, and be injected into a simulation with the use of a ghost. Ghosts are similar in most aspects to actors, but with respect to the simulation environment, ghosts are more efficient for producing the desired domain effects of a simulation. The script function of a ghost can be programmed to dispatch a notification message to the various

actors involved in a particular event scenario. In this way, many different simulations can be implemented by simply supplying a ghost with varying event scenarios for any given domain configuration.

Each actor and ghost participating in a given simulation is assigned a unique *stagename* which serves as an address for receiving communications from other characters. The messages received by a character are collected by SIMULACT and placed into each characters respective *mailbox.* It is the responsibility of every actor to check its mailbox and handle any messages it may find there in an appropriate and timely manner. SIMULACT provides three types of *mail objects* called *memos, futures,* and *future streams.* The most basic type of communication among the three, is the memo. A memo is a simple one-way communication from one actor or ghost to another actor. There is no obligation on the part of an actor which has received a memo to initiate a response. For instances when a response is both required and expected, an actor may choose to use the future mail object which has been especially provided for this purpose.

When using memos, an actor responding to a message, must actually send another memo containing the response. With this method, some care must be taken to ensure the actor receiving the reply will understand the message is in response to a memo sent previously. Using futures in these instances would greatly simplify the exchange of a reply. An actor which sends a future mail object actually holds onto a copy of it while another copy is sent to the desired actor. A response is all that is necessary for the actor to reply to this future. The actor's response will automatically be rerouted back to the sending actor's copy of the future by SIMULACT. In this way, the actor always knows which message the response is in reference to. Future streams are much like futures except the reply communication path remains open enabling the receiving actor to reply with multiple responses, as they are formulated, rather than sending them all at once.

3.4.2 An Overview of TESS

3.4.2.1 A Production System

For many areas of application requiring both an effective and practical means of producing "intelligent" problem solving behavior, the use of an *expert system* has emerged as the predominant choice. In essence, an expert system constitutes both the knowledge and skill which a human expert would possess. The type of program most often used to implement an expert system is called a *production system.* In a production system, the knowledge of a human expert is embodied in a collection of if-then type statements called *production rules,* or simply *productions.* Each production rule is used to assert a piece of knowledge in the form of a pairing of conditions and actions such that; if the conditions are satisfied (or TRUE), then the associated actions are appropriate. The condition and action parts

of a rule are commonly called its left-hand side (LHS) and right-hand side (RHS) respectively. Production systems employ an iterative control structure called an *interpreter* which enables an expert system to replicate the problem solving skills of a human expert. This is accomplished by repeatedly applying the relevant knowledge, expressed as production rules, at every stage in the problem solving process. If the expert system has been properly designed, it will incrementally converge upon a solution and eventually terminate.

As an example of how particular domain knowledge may be captured using production rules, consider the expertise of an automobile mechanic. Suppose a mechanic is confronted with the problem of a car whose engine "turns over", but will not start. Among the many pieces of knowledge used by the mechanic, two such pieces may be represented by the following rules; 1) *"IF the engine will not start AND there is gas in the gas tank, THEN check the fuel pump"*, and 2) *"IF the engine will not start AND there is gas in the gas tank, THEN check the ignition."* In both of these rules, the mechanic is predicating the actions of checking the fuel pump and ignition upon the simultaneous existence of two pieces of data, namely *the engine will not start* and *there is gas in the gas tank.* In a production system, data of this kind is stored in a global data base called *working memory.* The individual data stored in working memory is referred to as *working memory elements* (wmes), or more casually as *facts.* When a combination of facts stored in working memory satisfies the conditions stated in the LHS of a rule, they are "tagged" as constituting an *instance* of the particular rule. For example, if the two facts *the engine will not start* and *there is gas in the gas tank* were stored in working memory, they would be found to satisfy the conditions of rules one and two, and would be registered accordingly as representing an instance of each rule.

Every rule instance represents one combination of facts for which the actions of the corresponding rule are applicable given the current contents of working memory. In terms of the automobile mechanic, an instance at a particular stage in the problem solving process represents one possible action taken in order to isolate the reason for the engine's failure to start. As in the above example, there may be any number of instances generated as a result of the current contents of working memory. These instances represent a dilemma for the production system. Typically, all the associated actions *could* be performed at the same time, however, only one action may be executed at a time. To resolve this dilemma, all the instances generated are grouped together to form a *conflict set.* The production system then uses a process called *conflict resolution* to select and remove a single instance from the conflict set whose associated actions are to be performed. When the actions of the selected rule instance are executed, a process known as *firing* the rule, the action procedure is given access to the group of facts which constitute the instance. This allows the actions of a rule to manipulate the set of facts which satisfied its conditions. In particular, the firing of a rule can modify and delete these facts from working memory as well as create new facts to be added to working memory. Any changes made to the working memory will generally cause instances affected by these changes to be removed from the conflict set as well as

cause new instances to be added.

This cycle of recognizing rule instances and performing selected actions is commonly known as the *recognize-act* cycle and can be summarized as follows:

1. *Recognize*: Given the current contents of working memory, determine every possible way the LHS of each production rule can be satisfied.
2. *Conflict Resolution*: Select a single rule instance from among those in the conflict set. If there are none, then halt the interpreter.
3. *Act*: Perform the actions specified by the RHS of the selected rule instance.
4. Goto 1.

TESS is the name of a SIMULACT support package which implements an expert system *shell* written in Common LISP [78]. An expert system shell essentially provides a generic language for specifying production rules and a command loop for controlling, interrogating, and tracing the actions and elements of the interpreter's environment.

3.4.2.2 The Production Language

The language[6] used to specify production rules in TESS, incorporates a number of features which allow the conditional parts of rules to be specified in terms of explicit *patterns* of working memory element *attributes*. A production rule LHS, its conditional part, consists of a number of *condition elements* which represent the necessary conditions under which the associated actions of the rule may be applied. The set of condition elements comprising a rule LHS serve as an outline for a general structure, or pattern, of wmes which would satisfy its applicability requirements. Such patterns describe certain characteristics which all acceptable groups of wmes must exhibit. These characteristics are; 1) the number of wmes (not necessarily distinct) in the group, 2) the features which individual wmes are required to have (intra-element features), and 3) the features which the group as a whole must have (inter-element features). For each production rule, and for each corresponding group of wmes which are found to exhibit the specified pattern, an instance of the rule is created based on those wmes and is made a member of the conflict set.

Production rules are defined in TESS with a function[7] P (an abbreviation for production) which has the following general form:

[6] The production language designed for TESS was, in part, closely modeled after the production language of OPS5 [5] and consequently shares many of the same features. In fact, with the exception of the symbol used for the index operator, the language used by TESS to specify a production rule LHS is a superset of OPS5.

[7] However, in the actual Common LISP implementation, P is defined as a macro in order to avoid the interpreter evaluating any of its parameters. After essentially "quoting" these parameters, they are then passed to a function called P-1.

(P *rule-name* [*salience*]
(*condition-element* C0)
⋮
(*condition-element* Cn)
- ->
action-body)

The first parameter, *rule-name*, specified for each production rule must be a symbol. This symbol is never evaluated (in the LISP sense) and will be used to denote the name of the rule. Every rule name should be distinct from that of any other rule which has previously been defined. There are several reasons why this should be so. For instance, duplicate names would cause the interpretation of a rule firing trace to be ambiguous, thus making it that much more difficult to "debug" a program. But, perhaps the most important reason has to do with the method by which the actions of a rule are stored. The actions associated with each rule, as defined by their *action-body*, are stored after some pre-processing as an entry of a hash table for rapid retrieval. In defining a rule which has the same name as a previous one, the associated action of the previous rule would essentially be re-defined and may thus be the cause of unintended program behavior. TESS will issue a warning message in such instances to notify the user of the possibility for adverse side-effects. Production rules may also have an optional second parameter, as indicated by the enclosing square brackets, called *salience*. The salience parameter is used by a conflict resolution strategy as an aid in choosing the next rule instance to fire. Its value can be any numerical value, positive or negative, which establishes the relative priority of the rule in relation to others. By default, the value of the salience parameter is set equal to zero.

For the purpose of discussion, the condition elements specified in the LHS of a production rule will be referred to as C0, C1, C2, ect., where the condition element referred to as C0 is the first one specified, C1 is the second, and so on. The pattern of wme attributes described by the condition elements consists of various constraints on the allowable values of certain attributes. These constraints are expressed in the form of *attribute-value* pairs. Each attribute-value pair imposes a single constraint on the permissible values of a particular wme attribute. This constraint requires the attribute must relate to the paired value in a specific way. This relationship, unless explicitly stated otherwise, is by default, one of equality—the specified attribute of an attribute-value pair must be equal to the corresponding value. For example, an attribute- value pair which specifies an attribute AGE and a corresponding value 21, asserts a constraint which requires a wme to have an AGE attribute which is equal to the value 21. Constraints of this kind, involving only one wme, are actually specifications of characteristic intra-element features which each element of the working memory can be tested for on an individual basis. Typically, a wme will be recognized as exhibiting a number of the attribute patterns which are specified by the condition elements of various production rules.

Generally speaking, the patterns of wme attributes described by the condition elements of a typical production rule, are not so ingenuous as to require testing for intra-element features only. On the contrary, a typical production rule will describe a pattern of wme attributes which involves an inter-dependence of their values such that testing for inter-element features is also necessary. As an example, in a particular production rule, it is known the value of an attribute ATTR2 specified in condition element C2 must always equal the value of attribute ATTR1 specified in condition element C1. Situations of this type often arise in practice and are easily accommodated using a *variable* to establish a *conjunctive* constraint between the values of each attribute. In TESS, symbols which are used to identify variables are distinguished from ordinary symbols by enclosing them in angle brackets with no spaces inserted, as in the symbol <VAR>. By specifying a common variable, such as <VAR>, for the value attributes ATTR1 and ATTR2 in condition elements C1 and C2 respectively, a conjunctive constraint between the condition elements is defined. In particular, this constraint requires the value of attribute ATTR1 of condition element C1, by default, to be equal to the value of attribute ATTR2 of condition element C2. Conjunctive contraints specify tests for inter-element features which a group of wmes must exhibit in addition to any individual intra-element features. Only those groups of wmes, which have attributes agreeable with both kinds of features defined by a particular production rule, will be used to form specific instances.

3.4.2.2.1 Literals and Working Memory Element Attributes Thus far, nothing has been said regarding the association of various attributes with particular working memory elements. Working memory elements are generally considered to represent various *objects* such as automobiles, houses, plants, animals, etc. The kinds of objects wmes can represent is not limited to merely tangible objects but, they can also represent many kinds of abstract objects such as, control knowledge, statistical data, concepts, and so forth. Wmes can be used to represent almost any kind of object. In addition, it can be said of all these objects thate each has the following aspects in common: 1) each object belongs to a particular class, and 2) within each class an individual object is distinguished from the others by its particular attributes. For example, these aspects are most easily seen when considering objects belonging to a class named PERSON. Each person has many attributes which are frequently used to distinguish one from another, such as their physical attributes of height, weight, sex, eye color, etc., as well as other attributes such as their social security number, date of birth, address, telephone number and so on. All of these attributes, and perhaps more which were not mentioned, would likely be associated with each objects in the class labelled PERSON. In TESS, the mechanism for declaring a class of objects and the names of attributes which are to be associated with those objects is a (macro-) function called LITERALIZE whose general form is as follows:

(LITERALIZE *object-class*
object-attribute 0
⋮
object-attribute n)

A LITERALIZE statement is used to establish an *association* between the names of object attributes (literals) affiliated with a class of objects and individual storage locations in the underlying representation of wmes. Working memory elements are represented internally with an addressable data structure, or to be more precise, a Common LISP sequence. Each wme sequence is uniform in length.[8] A Common LISP sequence is addressable in the same sense that arrays are normally thought to be. Each sequence is a contiguous block of memory locations, the individual contents of which can be efficiently accessed with an appropriate index. Using LISP sequences adds the advantage that they may also be treated as ordinary list structures thereby admitting all the usual list operations. By convention, the elements in a sequence of length n are addressed with indexes $0, 1, 2, \ldots, n-1$. With respect to object representation, each element of a wme sequence is considered to contain the value of an object attribute. The LITERALIZE statement essentially gives names to some of these object attributes, which, in effect lends a certain interpretation to the contents of a wme. Internally, TESS reserves the first attribute of each wme, the value stored in address 0, and interprets it as a special *time stamp* attribute. A time stamp is a positive integer which reflects the time, in terms of a relative ordering, at which a particular wme was created (i.e. added to the contents of the working memory). Each successive time stamp is one number larger than the previous, and no two time stamps have the same value. One way in which time stamps are used is in referring to individual wmes. For example, when examining the contents of working memory, a wme with a time stamp attribute of 15, for instance, would be designated as element F15. This allows a user to specify a particular wme to be removed from the contents of working memory.

[8] In the current implementation of TESS, the length of a sequence used to represent a wme is fixed by a system parameter to 32 elements. This value allows classes of objects to be defined with as many as 30 user-defined attributes. If a program's requirements demand greater numbers of attributes, the parameter can be changed to accommodate as many attributes as are needed and the system is then re-compiled.

The procedure by which LITERALIZE statements associate the names of object attributes with specific wme attributes is best explained with the use of an example. With this in mind, consider the following set of statements abstracted from an implementation of the "Monkey and Bananas"[9] problem:

```
(LITERALIZE PHYS-OBJECT NAME AT WEIGHT ON)

(LITERALIZE MONKEY AT ON HOLDS)

(LITERALIZE GOAL STATUS TYPE OBJECT-NAME TO)
```

In the "Monkey and Bananas" problem, there is a monkey in a cage with various objects such as a couch, ladder, and several boxes. In the center of the cage hangs a bunch of bananas. Unfortunately, the bananas are much too high for the monkey to reach them. However, if the monkey is able to reason out a plan of action which utilizes other items in its cage, such as the ladder, it would be possible for the monkey to reach the bananas and eat them. The couch, ladder, boxes, and so on, in the monkey's cage are represented as specific items in a class of objects called PHYS-OBJECT. Their attributes, from the first LITERALIZE statement above, are NAME (what the object is), AT (Cartesian coordinates representing location of object), WEIGHT (relative weight of object such as heavy, light, etc.), and ON (what the object is resting on). Similarly, the monkey's present activity is represented by objects in a class called MONKEY which has the attributes of AT, ON, and HOLDS (object monkey is holding). Notice the existence of duplicate attributes (i.e. AT and ON) defined for both object classes PHYS-OBJECT and MONKEY. The stages of the monkey's planning activities are represented by objects belonging to a GOAL class of objects. Their attributes are STATUS (either active or satisfied), TYPE (action to be performed by the monkey), OBJECT-NAME, and TO (location to move the object to).

What results from mapping the names of object attributes onto the indices of specific wme attributes, in some sense may be considered a kind of template for interpreting the contents of a wme. Such templates, when used to "screen" wmes, filter their components in such a way that only those which correspond to the attributes of a certain object class are meaningful. This interpretation of the purpose of literal mapping coincides with the way in which condition elements are normally specified, that is, with a structure much like that of predicates in first-order logic. A specially named object attribute is defined internally as CLASS, which can be used to facilitate this interpretation. The CLASS literal always maps to the first available index of each template, address 1 of a wme. The class of object which a wme represents, is normally stored in this location. For example, a wme which represents a particular physical object in the "Monkey and Bananas" problem would have the name of its object class, PHYS-OBJECT, stored in attribute 1 of the wme sequence. In this way,

[9] The implementation of the "Monkey and Bananas" problem used for demonstrative purposes in this chapter was adapted with few modifications from [5] Appendix 1, pp. 383–408.

the class of an object can be treated as just another wme attribute. The names of attributes associated with a class of objects, as defined by a LITERALIZE statement, are then mapped in the order they were specified onto the remaining wme attributes beginning with the first available address, which is the index 2. The result is the following correspondence between the literals of a template for the PHYS-OBJECT class of objects and wme attributes.

Wme Attributes:	0	1	2	3	4	5	6	···
Template:	*	CLASS	NAME	AT	WEIGHT	ON	—	—

While a template representation of the attributes associated with an object class is only a conceptual contrivance, when expressed visually (as in the above example) the notion of interpreting the contents of a wme can be seen more clearly. For instance, if the value stored in the CLASS attribute of a wme is PHYS-OBJECT, then, with respect to the above template, the values stored in indices 2, 3, 4, and 5 of the wme have a definite interpretation. In particular, the contents of each wme index has the interpretation which is indicated by the corresponding template literal. On the other hand, if the value stored in a wme's CLASS attribute is *not* PHYS-OBJECT, then the mapping of literals represented by the above template *could* be entirely without meaning. However, there is the possibility, that a number of the literals present in the template for a PHYS-OBJECT class of object are the same as those in the templates of other classes of objects. When this situation occurs, as it does with the ON and AT literals for objects of class PHYS-OBJECT and MONKEY, the indices associated with subsequent occurrences of previously encountered literals are retained. In other words, a particular attribute literal is mapped to a specific wme index only once. Any future LITERALIZE statements which names a previously mapped attribute literal, must accommodate the earlier mapping in their object class "templates". The purpose of handling duplicate object attribute literals in this manner is to make it possible for production rules to treat the attributes of objects, within different classes, in a more generalized way.

The process of literal mapping can now be continued for the MONKEY and GOAL object classes. Including the internally defined CLASS attribute literal, the previously mapped literals of the MONKEY class are; CLASS = 1, AT = 3, and ON = 5. Viewing these associations as a partial description for a MONKEY object template, the index corresponding to the first available wme attribute is index 2. This is the wme index which the remaining attribute literal, HOLDS, is associated with. Notice how this mapping leaves a hole in the MONKEY object template at wme index 4 as there is no additional attribute associated with a MONKEY object which would occupy that location. Holes in object attribute templates are common and are of little consequence. Like the object attributes defined for the PHYS-OBJECT object class, none of the attributes defined for the GOAL object class had been previously mapped. The mapping procedure for its attribute literals is a straightforward one. The final results of the attribute literal mapping procedure for the LITERALIZE statements given above, are summarized in Table 3.1. The entries

Object Attribute Literals	Object Classes		
	PHYS-OBJECT	MONKEY	GOAL
CLASS	[1]	[1]	[1]
NAME	2		
AT	3	[3]	
WEIGHT	4		
ON	5	[5]	
HOLDS		2	
STATUS			2
TYPE			3
OBJECT-VALUE			4
TO			5

Table 3.1: Association of Attribute Literals with Wme Indices.

of Table 3.1 are the wme index values to which the various object attribute literals were assigned. Those entries which appear in a box indicate the particular attribute literal was associated with a wme index in a LITERALIZE statement.

3.4.2.2.2 Pattern Specification Syntax In previous discussions, it was described on an elementary level, how patterns of wme attributes are specified by the condition elements of production rules. Recall that the detection of such patterns, in terms of identifying suitable groups of wmes, typically involves testing for both intra- and inter-element group features. These features are specified in a production rule's condition elements with a series of attribute-value pairs. Because the exact nature of wme attributes had not yet been introduced, the actual means by which attribute-value pairs are specified was not discussed. However, now that the association between object attributes names and wme index locations has been discussed, the specification of attribute-value pairs can now be presented. The *index operator*, a special operator with the exclamation mark (!) as its symbol, is used to identify a particular wme attribute in terms of its index address. The index operator can be applied to either a positive integer, or an object attribute literal. Applying the index operator to an integer or literal is accomplished by placing it directly in front of them—there may or may not be any space in between. In the case the index operator is applied to an integer, the wme attribute specified is the one which corresponds to the index of the same number. For example, when the index operator is applied to the integer 5, as in !5, the wme attribute which is specified is the one occupying index address 5.[10] When the index operator is applied to a literal however, the specified wme attribute

[10] Note that the index operator could also be used to gain access to a wme's time stamp attribute, as in !0. Although it is not clear what purpose this may serve.

corresponds to the index which the literal was associated with by a previous LITERALIZE statement.[11] For example, applying the index operator to the internally defined attribute literal CLASS, as in !CLASS, specifies the wme attribute which occupies index address 1, normally the name of the object's class. Following the specification of each attribute is a corresponding value of an attribute- value pair. The specified wme attribute is required to have a certain relationship to this value, by default, they must equal. A condition element in a production rule for the "Monkey and Bananas" problem might look like the following:

```
(!CLASS PHYS-OBJECT   !NAME LADDER   !AT <P>   !ON FLOOR)
```

Extra space has been added between each attribute-value pair for emphasis. With respect to the association of literals and wme indices summarized in Table 3.1, the above condition element could also have been written as:

```
(!1 PHYS-OBJECT   !2 LADDER   !3 <P>   !5 FLOOR)
```

The pattern constraints described by each of the above condition elements will only be satisfied by wmes which have the following intra-element features: 1) the value stored in attribute 1 is PHYS-OBJECT, 2) the value stored in attribute 2 is LADDER, and 3) the value stored in attribute 5 is FLOOR. Since the specified value for attribute 3 is a variable, and there are no other condition elements to impose a conjunctive pattern constraint, wme attribute 3 may contain any value. This is also true of all wme attributes which are not constrained by the condition element.

When the contents of a condition element are processed, to determine what pattern constraints have been specified, a counter is maintained internally which steps through each wme index to be addressed. For each condition element, the initial value of the counter is set to 1, which reflects the index address normally associated with a wme's CLASS attribute. As successive values are specified in the condition element, the counter is incremented to reference the next consecutive wme attribute. In actuality, the attribute specified in an attribute-value pair is really determined by the current contents of the index counter and *only* values are specified in condition elements. Consider, for example, the attribute-value pairs which are specified by the following condition element:

Condition Element:	(ARG1	ARG2	ARG3)
Index Counter:	1	2	3

The values of ARG1, ARG2, and ARG3, in conjunction with the current contents of the index counter, specify the following attribute-value pairs:

[11] It is for this reason that all LITERALIZE statements in a production system declaration file should be located before the definition of the first production rule. In this way, all literals will have been associated with a wme index before the index operator is first applied.

Attribute 1 = ARG1
Attribute 2 = ARG2
Attribute 3 = ARG3

With the use of an index operator, be it applied to a positive integer or attribute literal, the progressive sequence of wme indices normally produced by the index counter can be arbitrarily altered. During the processing of a condition element, when an index operator is encountered, it is treated as a *directive* which specifies a wme attribute to be stored in the index counter. In particular, the new wme attribute is the one which corresponds to what the operator is being applied to. Once the index counter has been set to this new value, its contents, as before, are incremented for each value specified in a condition element. For example, consider the following condition element:

Condition Element:	`(ARG1`	`!5 ARG2`	`ARG3)`
Index Counter:	1	5	6

The presence of the directive !5, sets the value stored in the index counter to 5 which is the wme attribute used for the next encountered value. Following are the attribute-value pairs which are specified by the above condition element:

Attribute 1 = ARG1
Attribute 5 = ARG2
Attribute 6 = ARG3

In the preceding examples, the relation between wme attributes and specified values which needed to be satisfied was, by default, taken to be equality. That is, it was necessary for a wme attribute to be equal to the corresponding value specified in the respective condition element. However, any one of a number of relations could have optionally been selected as the one to be used with each specified value. The need for a particular relation between an attribute and value to be satisfied is expressed in a condition element by placing a special relation symbol immediately before the specified value. The symbol '<', for example, is used to denote the "less than" relation. When this symbol is placed in front of a value, for instance as in < 35, it specifies that a suitable value for a wme attribute is any number, so long as it is *less than* 35. Table 3.2 gives a complete summary of the attribute-value relation symbols and their meanings which are available in TESS. Consider the following example which incorporates many of the production language aspects discussed up to this point;

CE:	`(ARG1`	`!5 > 21`	`!5 <= 65`	`ARG2`	`!3 # (RED BLU GRN))`
Index:	1	5	5	6	3

The attribute-value pairs specified by the condition element are summarized in the following:

Attribute 1	=	ARG1
Attribute 3	#	(RED BLU GRN)
Attribute 5	>	21
Attribute 5	<=	65
Attribute 6	=	ARG2

Notice how the index operator was used to impose *two* pattern constraints on the value of wme attribute 5. The first attribute-value pair involving attribute 5 requires its value to be *greater than* 21, while the second requires its value to be *less than or equal to* 65. Together, they specify a *conjunction* of intra-element feature tests which are to be applied to a single wme attribute. In particular, they specify an interval of suitable attribute values, such that, if x is the value of wme attribute 5, then it must lie somewhere within the limits defined by $21 > x \geq 65$. In general, a conjunction may consist of any number of pattern constraints. Although conjunctions could be defined as in the above condition element, a special syntax is provided to emphasize the presence of such conjunctive attribute-value constraints. This syntax has the following general form:

$$\{[< variable >] \quad [[[relation1] \ value1] \quad \cdots \quad [relationN] \ valueN]\}$$

where bracketed items are optional. The first parameter is an optional variable which, if present must be the first occurrence within a production rule, the particular variable name has been used. When this variable is present, it refers to the value of the particular wme attribute (i.e. it is "bound" to the value of the attribute), and as such, it does not represent an attribute-value pair. The remaining (optional) parameters are values which represent the conjunctive attribute-value constraints. If relational symbols are present, they pertain to the immediately following value. If a relation is not specified for a particular value, equality is used by default. Using this syntax, the conjunction of values for wme attribute 5 in the above condition element could have also been specified as:

$$\{> 21 <= 65\}$$

Like the special syntax which can be used to denote a conjunction of attribute values, there is also an alternate representation for a *disjunction* of attribute values. A disjunction of values for a particular wme attribute is indicated by enclosing a number of "literal" values within two special symbols, << ␣ and ␣ >>, where ␣ is used here to represent a required space character. The value of the particular wme attribute may be *equal to* any one of the enclosed literal values. Because enclosed values are taken literally, a symbol, which under normal circumstances would be taken as a variable, can be specified as one of the disjunctive values. In the example condition element given above, the value and relation specified for wme attribute 3 is essentially the same as a disjunction. The specified value is a list containing three symbols, RED, BLU, and GRN. The relation symbol, #, indicates

A < B	Wme attribute A is less than the value of B. Can be used for arithmetic comparisons as well as lexicographic string comparisons (as with **STRING<** function).
A > B	Wme attribute A is greater than the value of B. Can be used for arithmetic comparisons as well as lexicographic string comparisons (as with **STRING>** function).
A = B	Wme attribute A is equal to the value of B. Can be used for arithmetic, symbol, string, and general LISP data structure comparisons.
A @ B	Wme attribute A is a list and has the value of B as a member. The value of B can be any LISP object (the **EQUAL** function is used).
A # B	Wme attribute A is a member of the list which is the value B. The attribute A can be any LISP object (the **EQUAL** function is used).
A <> B	Wme attribute A is not equal to the value of B. Can be used for arithmetic, symbol, string, and general LISP data structure comparisons.
A <= B	Wme attribute A is less than or equal to the value of B. Can be used for arithmetic comparisons as well as lexicographic string comparisons (as with **STRING<=** function).
A >= B	Wme attribute A is greater than or equal to the value of B. Can be used for arithmetic comparisons as well as lexicographic string comparisons (as with **STRING>=** function).
A ~@ B	Wme attribute A is a list and does not have the value of B as a member. The value of B can be any LISP object (the **EQUAL** function is used).
A ~# B	Wme attribute A is not a member of the list which is the value B. The attribute A can be any LISP object (the **EQUAL** function is used).
A <=> B	Wme attribute A is of the same data type as the value of B. Can be used for any LISP objects suitable for comparison with the **TYPE** function.

Table 3.2: Attribute-Value Relation Symbols Available in TESS.

that wme attribute 3 should be tested for membership in this list (refer to Table 3.2). Therefore, the values for which attribute 3 of a wme are acceptable can be one of the symbols RED, BLU, or GRN. Combining the alternate syntactic representations for both a conjunction and disjunction of attribute values, the previous example condition element is entirely equivalent to the following:

```
(ARG1  !5 {> 21  <= 65}  ARG2  !3 << RED BLU GRN >>)
```

Several other features of the TESS production language have yet to be discussed and can be easily described by considering an example production rule. The following production rule, abstracted from an implementation of the "Monkey and Bananas" problem, will be used to identify the remaining production language features:

```
(P HOLDS-OBJECT-CEIL
  {(GOAL !STATUS ACTIVE !TYPE HOLDS !OBJECT-NAME <O>)          <GOAL>}
  {(PHYS-OBJECT !NAME <O> !WEIGHT LIGHT !AT <P> !ON CEILING) <OBJECT>}
   (PHYS-OBJECT !NAME LADDER !AT <P> !ON FLOOR)
  {(MONKEY !ON LADDER !HOLDS NIL)                             <MONKEY>}
 - (PHYS-OBJECT !ON <O>)
-->
   (FORMAT T "~2%Grab ~A~%" <O>)
   (MODIFY <MONKEY> !HOLDS <O>)
   (MODIFY <OBJECT> !ON NIL)
   (MODIFY <GOAL> !STATUS SATISFIED))
```

Perhaps the most obvious language feature which has yet to be discussed is the use of a minus sign in front of a condition element, as with the fifth condition element, C4. A minus sign is put in front of any condition element describing wme attributes in which it is required that *no* element currently in the working memory satisfies. With respect to the above rule, condition element C4 requires there to be no wme present in the working memory which is of class PHYS-OBJECT and has an !ON attribute equal to the value specified by variable <O> (i.e. the contents of attribute !OBJECT-NAME of a wme corresponding to condition element C0). In terms of a suitable group of wmes which exhibit the described pattern of wme attributes, a total of four satisfactory wmes are required to exist simultaneously in the current contents of the working memory while a fifth must not exist. Thus, for each condition element which is *negated* by a preceding minus sign, a group of wmes which is one less wme in size than the total number of condition elements is needed to satisfy and instantiate, a production rule. A second notable language feature is the presence of braces enclosing certain condition elements, in particular condition elements C0, C1, and C3. Located outside each of these condition elements, inside the braces and to the far right, is a variable symbol. The braces are used to associate or "bind", the variable to the particular wme of a rule instance which corresponds to the condition element. This is the

basic method by which the action-body of a production rule can reference a particular wme of a group which constitutes an instance of the rule. The variable can appear on either side of a condition element so long as it is inside the braces. The names of such variables should also be unique and distinct from those used in any of a rule's condition elements. More will be said concerning the internal use of these variables in the action-body of a production rule in the next section.

A production language feature not demonstrated by the above rule comes into play when a rule fails to specify any condition elements. Intuitively, this means that whatever actions may have been programmed into the rule, they are applicable no matter what the contents of the working memory may be. Indeed, they are applicable even when there are no elements in the working memory. To insure something is present in working memory to match against a production rule with no specified condition elements, a technique was borrowed from the CLIPS [33] expert system shell. In CLIPS, when rules are processed in which there are no specified "condition elements", a special pattern is *inserted* into the rule's definition. This pattern always matches an "initial-fact" which is automatically asserted and placed into the working memory when the system is "reset". In this way, from a programming standpoint, there is something in the working memory which will match and consequently instantiate a production rule which has no specified condition elements.

3.4.2.2.3 Production Rule Action-Bodies and Lambda-Expressions Due to the fact that TESS is both implemented in Common LISP and intended to be executed in a "standard" LISP environment, there are few restrictions which limit the kinds of actions that may comprise a production rule action-body. The actions associated with a production rule are actually nothing more than a succession of ordinary LISP function calls which are written as if they were the body of another function. In fact, when each production rule is processed, its associated actions are made to be the body of a lambda-expression [78, pages 75–83] which requires a single functional argument. With respect to a particular production rule, the argument given to its corresponding lambda-expression is a list with elements representing a group of wmes which constitute an instance. The group of wmes in the list are ordered with respect to the condition element which they were found to satisfy. Thus, the execution of a production system proceeds by successively applying, in the LISP sense, lambda-expressions to corresponding rule instances of the conflict set. In order to facilitate this kind of execution, the instance data structures which comprise the conflict set supply the hash table key (i.e. rule name) by which the appropriate lambda-expression is retrieved. This lambda-expression is then applied to the group of wmes "stored" in the instance data structure.

As an example of how a lambda-expression is constructed for a particular production rule, consider, once again, the HOLDS-OBJECT-CEIL rule described in the previous section. In all, there are a total of five different variables used in the rule, the use of which

Variable	Type	Binding Address	Used in Action-Body
<O>	ATTR	C0(!OBJECT-NAME)	×
<GOAL>	WME	C0	×
<P>	ATTR	C1(!AT)	
<OBJECT>	WME	C1	×
<MONKEY>	WME	C3	×

Table 3.3: Variables Used in HOLDS-OBJECT-CEIL.

can be classified into one of two categories; 1) variables which refer to a particular wme attribute (ATTR), and 2) variables which refer to an entire wme (WME). With respect to the components of a production rule instance, each variable can be identified with a particular constituent wme and/or attribute. In the implementation of production rules, variables are nothing more than symbols which have been associated with instance components, they are not bound in the conventional LISP sense. However, because the action-body of a production rule is used as the body of a lambda-expression, a variable symbol which appears in the action-body will, when executed, be evaluated in the LISP environment as if it were bound to some value. This disparity between the way in which production rule variables are used internally for pattern matching purposes, versus their use in an action-body, is compensated for by including a *header* in the lambda-expression created for each rule. A header provides LISP code sufficient to encapsulate an action-body in a local execution environment. Within this environment, bindings are provided for the variable symbols which appear in a rule's action-body thereby allowing them to be evaluated directly by the LISP interpreter. Each variable symbol is made to point to the component of an instance which corresponds to the value specified in the condition elements of the respective production rule.

In relation to an action-body, the important aspect of each variable is the *address* to which it is a reference, regardless of what the address may contain. This address can either be a wme attribute index address or a location within an instance, as supplied to a lambda-expression, where a particular wme can be found. Table 3.3 summarizes the address components referenced by the variables used in the HOLDS-OBJECT-CEIL production rule. This information will be encoded into the header of the corresponding lambda-expression. The variables which are listed in the table as being of type WME have as their specified binding address the condition element which they reference. For example, in the particular production rule, the variable <MONKEY> is bound to the condition element referred to as C3, which is used as the binding address in the Table. What this means is that when a function call in the action-body of the HOLDS-OBJECT-CEIL rule refers to the variable <MONKEY> in the corresponding lambda-expression, it can be treated as a reference to element 3 (the fourth component) of the wme instance list argument.

The variables in Table 3.3 which are listed as being of type ATTR and which occur in the action-body of HOLDS-OBJECT-CEIL, can be treated in the corresponding lambda-expression in a manner similar to the WME type variables. However, with ATTR variables, two addresses are required to access the particular wme attribute. The first address is a reference to the condition element in which the variable was bound. In particular, this is the condition element in which the variable is first specified under the equality relation as the value of a specified wme attribute. The index corresponding to the specified wme attribute is used as the second address. For example, the variable <O> in the HOLDS-OBJECT-CEIL production rule is bound in condition element C0 as the specified value of a wme attribute OBJECT-NAME. In Table 3.3, the complete binding address is specified as C0(!OBJECT-NAME) which is used to connote the need for two separate address components. With respect to variable references within the body of a corresponding lambda-expression, the C0 address component is an indication that the value actually being referenced is contained in the first element of the supplied wme instance list argument. The second address component, !OBJECT-NAME, specifies which wme attribute of that first element is the referenced value. The address information contained in Table 3.3 is used when processing the HOLDS-OBJECT-CEIL production rule to create the header LISP code for the corresponding lambda-expression. This is done using a LET special-form which encloses the production rule's action-body. For example, the following special-form is used in the lambda-expression created for the HOLDS-OBJECT-CEIL rule:

```
(LET ((<O>       (ELT (ELT *FACTS* 0) 4))
      (<GOAL>          (ELT *FACTS* 0))
      (<OBJECT>        (ELT *FACTS* 1))
      (<MONKEY>        (ELT *FACTS* 3)))
         --- Action-Body ---            )
```

The symbol *FACTS* is a variable argument specified by each lambda-expression which a wme instance list is bound to when the function is applied.

There are a number of standard (macro-) functions defined in TESS which are especially designed for manipulating wmes in various ways. The two most commonly used functions are named ASSERT*[12] and RETRACT. When wmes are *asserted*, with an ASSERT* statement, the wmes are added to the contents of working memory. Likewise, a *retraction* of wmes, by a RETRACT statement, removes the wmes from the contents of working memory. The general form of the ASSERT* and RETRACT statements are given below:

[12] A function called ASSERT is already a part of the Common LISP function repertoire. In order to avoid re-defining the function, an asterisk was appended to the name.

(ASSERT* (*attribute-value list*0)
(*attribute-value list*1)
⋮
(*attribute-value listn*))

(RETRACT *wme*0 *wme*1 ··· *wmen*).

Any number of new wmes can be added to the contents of working memory with an ASSERT* statement. The values of each wme's attributes are specified as a list of attribute-value pairs in exactly the same way that patterns are specified in the condition elements of a production rule. The only exception is that all attribute values must be specified under the equality relation. For example, the following ASSERT* statement when executed would assert two new wmes whose attributes have the indicated values:

(ASSERT* (PHYS-OBJECT !NAME <O> !WEIGHT LIGHT !AT X5-Y3)
(GOAL !STATUS ACTIVE !TYPE ON !OBJECT-NAME <O>))

The ASSERT* function is also the mechanism used at the top-level of TESS (i.e. the shell environment) for adding new wmes to the contents of working memory. However, the above example could not be processed at the top-level because it specifies a variable, <O>, as the value of a wme attribute. This can only be done within the action-body of a production rule where the variable is a reference to a specific value. Like ASSERT*, the RETRACT function is also the mechanism used to remove wmes from working memory at the top-level. When used at the top-level, the wmes to be removed are specified with respect to their time stamp attributes. On the other hand, when used in the action-body of a production rule, variables references to specific wmes of an instance are specified. To illustrate this distinction, consider the two example RETRACT statements given below:

(RETRACT 15 37 8)

(RETRACT <WME2> <FACT>).

The first statement is a typical example of how the RETRACT function would be used at the top-level. It instructs TESS to remove from working memory those wmes which have the time stamp attributes 15, 37, and 8. Incidentally, when RETRACT is used at the top-level, as in this example, it is not an error to have specified a time stamp for which there is no corresponding wme. If there were a wme in working memory with the specified time stamp, it would be removed anyway. The second statement above is an example of how the RETRACT function would be used in the action-body of a production rule. In this case, the variables <WME2> and <FACT> were presumably used in the LHS of the rule as

references to particular wmes of an instance. When used in the action-body of a production rule, it would be an error to non-existent wmes (i.e. to specify unbound variables).

Another standard function provided in TESS for manipulating wmes is the MODIFY (macro-) function. Unlike the ASSERT* and RETRACT functions which can be used in production rule action-bodies as well as at the top-level, the MODIFY function can only be used in action-bodies. In a sense, the MODIFY function may be considered to embody a combination of the operations performed by the ASSERT* and RETRACT functions. The MODIFY function is used to *modify* a number of wme attributes, where the particular wme is one of those which comprise a rule instance. Its operation proceeds as follows: first, a copy of the wme is made and the original is removed from the working memory; next, the necessary modifications are made to the attributes of the copy, and finally, the modified copy is added to the contents of the working memory. However, before the wme copy is asserted, an additional modification is made to its time stamp attribute. The new value stored in this attribute is a positive integer representing the latest wme time stamp. This is an important modification as it serves to distinguish between the old wme and the newly modified one. The general form of the MODIFY function is given below:

(MODIFY *wme-reference attribute-value-list*)

The *wme-reference* argument to the MODIFY function is a variable which, within a particular production rule, is "bound" to one of the condition elements. Following such a variable are the attribute-value pairs which are used to specify the wme attributes which are to be modified to contain the specified values. Like the attribute-value lists required for the ASSERT* function, the attribute-value list given to a MODIFY function must also be specified under the equality relation. The example HOLDS-OBJECT-CEIL production rule under consideration lists three separate MODIFY statements, the first of which is the following:

```
(MODIFY <MONKEY>  !HOLDS <O>)
```

In the LHS of rule HOLDS-OBJECT-CEIL, the variable <MONKEY> is "bound" to condition element C3 which is a requirement for ensuring the monkey is not presently holding anything (i.e. !HOLDS NIL). Also, variable <O> is "bound" to the name of the physical object which is the goal of the monkey to grab. The purpose of this MODIFY statement then, is to change the monkeys status to reflect a condition in which the monkey is currently holding the object. With respect to the creation of a lambda-expression which corresponds to the HOLDS-OBJECT-CEIL production rule, the above MODIFY statement, which is actually a macro call, is expanded before being inserted into the lambda-expression. The LISP code produced by this expansion, which is inserted verbatim into the lambda-expression, is shown below:

```
(LET ((*FACT* (COPY-SEQ <MONKEY>)))
```

```
(FUNCALL
  (SIMVAL 'TESS::*PROCESS-ASSERT-NEAR-HOOK*) *FACT* :RETRACT)
(SETF (ELT *FACT* 0) (FACT-TIME-STAMP)
      (ELT *FACT* 2) <O>)
(FUNCALL
  (SIMVAL 'TESS::*PROCESS-ASSERT-NEAR-HOOK*) *FACT* :ASSERT))
```

The procedure outlined for the operation of the MODIFY function, as explained previously, can easily be recognized in the above LISP code. A programming variable, *FACT*, is first set equal to a copy of the instance wme referenced by <MONKEY>. The function, pointed to by the global variable TESS::*PROCESS-ASSERT-NEAR-HOOK*, which is accessed in SIMULACT with the SIMVAL function, is then used to remove the wme from the working memory. After the copy's attributes have been modified, this same function is used to add it to the contents of the working memory. However, the actual altering of attribute values, is performed by the interposed SETF macro. Recalling that index address 0 is where the time stamp attribute of each wme is stored, it can be seen that the first attribute modified by SETF is the time stamp attribute, the value of which is returned by the FACT-TIME-STAMP function. Referring back to the previous MODIFY statement, the attribute to be modified is specified as !HOLDS, which from Table 3.1 maps to index address 2 of a wme. Therefore, the next attribute modified by SETF is the one located at index address 2. The value placed in this wme location is the value bound, by the lambda-expression, to the symbol <O>. LISP code of the same general form as above is inserted into the lambda-expression for each occurrence of a MODIFY statement. The lambda-expression created for the HOLDS-OBJECT-CEIL production rule is given below:

```
(LAMBDA (*FACTS*)
  (DECLARE (OPTIMIZE SPEED))
  (LET ((<O>       (ELT (ELT *FACTS* 0) 4))
        (<GOAL>          (ELT *FACTS* 0))
        (<OBJECT>        (ELT *FACTS* 1))
        (<MONKEY>        (ELT *FACTS* 3)))
    (FORMAT T "~2%Grab ~A~%" <O>)
    (LET ((*FACT* (COPY-SEQ <MONKEY>)))
      (FUNCALL
        (SIMVAL 'TESS::*PROCESS-ASSERT-NEAR-HOOK*) *FACT* :RETRACT)
      (SETF (ELT *FACT* 0) (FACT-TIME-STAMP)
            (ELT *FACT* 2) <O>)
      (FUNCALL
        (SIMVAL 'TESS::*PROCESS-ASSERT-NEAR-HOOK*) *FACT* :ASSERT))
    (LET ((*FACT* (COPY-SEQ <OBJECT>)))
      (FUNCALL
```

```
    (SIMVAL 'TESS::*PROCESS-ASSERT-NEAR-HOOK*) *FACT* :RETRACT)
  (SETF (ELT *FACT* 0) (FACT-TIME-STAMP)
        (ELT *FACT* 5) 'NIL)
  (FUNCALL
    (SIMVAL 'TESS::*PROCESS-ASSERT-NEAR-HOOK*) *FACT* :ASSERT))
(LET ((*FACT* (COPY-SEQ <GOAL>)))
  (FUNCALL
    (SIMVAL 'TESS::*PROCESS-ASSERT-NEAR-HOOK*) *FACT* :RETRACT)
  (SETF (ELT *FACT* 0) (FACT-TIME-STAMP)
        (ELT *FACT* 2) 'SATISFIED)
  (FUNCALL
    (SIMVAL 'TESS::*PROCESS-ASSERT-NEAR-HOOK*) *FACT* :ASSERT))))
```

Before the above lambda-expression is stored in a hash table as the action to be associated with the HOLDS-OBJECT-CEIL production rule, it is first compiled so that it can be executed more efficiently. Since production systems are relatively slow in comparison with other types of programming systems, a compiler feature which optimizes the resulting code for speed, is declared in each lambda-expression created. This declaration can be observed in the second line of the lambda-expression above.

3.4.3 The Pattern Matching Algorithm

An important component of any production system is the pattern matching algorithm employed by its interpreter. In a production system, the pattern matching algorithm is responsible for maintaining the correctness of the conflict set from one cycle of the interpreter to the next. The algorithm accounts for changes made to the working memory as the result of a rule firing, by making appropriate modifications (i.e. the addition and deletion of various rule instances) to the conflict set. Classical pattern matching algorithms accomplish this task by repeatedly computing the *entire* conflict set for each cycle of the interpreter. For every production rule LHS, the pattern matcher will iterate over all suitable combinations of working memory elements to find those which satisfy the specified conditions. When a satisfactory combination is found, an instance of the particular rule is created and added to the conflict set. Unfortunately, such simplistic algorithms are extremely inefficient and seriously reduce production system performance. On the other hand, the RETE [26] pattern matching algorithm has been specially designed to address the many pattern/many object pattern matching requirements of production systems. By using a special matching network, which saves certain state information, the RETE algorithm avoids iterating over the contents of working memory, as was previously done. Consequently, the RETE algorithm is able to achieve a significant increase in the overall performance of production systems and

has emerged as one of the most efficient pattern matching algorithms.[13]

The match network used in conjunction with the RETE algorithm is partitioned into two inter-connected networks called the *pattern network* and the *join network.* Their structure and inter-connections correspond to an aggregate representation of the conditions expressed within a collection of production rules. In essence, the match network combines the individual pattern matching requirements common to a number of production rules into a "single" comparison. The results of such a comparison are effectively distributed among its corresponding production rules, thus, the costs associated with redundant pattern matching are eliminated. The motivation for sharing such intermediate pattern matching results was due to observations of actual expert system implementations. It was found that in a typical expert system, the collection of production rules were organized into a number of logical groups, with each group containing various rules whose objectives are closely related. Among the rules comprising any particular group there was often found to be a considerable amount of structural similarity exhibited by their respective condition elements. Therefore, the pattern network was designed to share the results of comparisons for intra-element features among the collection of production rules. Likewise, the join network was designed to share the results of comparisons for inter-element features. As an example of how each network is structured, consider how the following two general english statements pertaining to the "Monkey and Bananas" problem might be realized.

"**IF** *the monkey is to get on some physical object* **AND**
the physical object is on the floor **AND**
the monkey **is** *standing next to it holding nothing,*
THEN
have the monkey climb onto the physical object."

"**IF** *the monkey is to get on some physical object* **AND**
the physical object is on the floor **AND**
the monkey **is not** *standing next to it,*
THEN
have the monkey first walk over to the location
of the physical object."

These two closely related english statements were coded in an actual implementation of the "Monkey and Bananas" problem using TESS in the form of two production rules named ON-PHYS-OBJECT and ON-PHYS-OBJECT-AT-MONKEY respectively. Their

[13]The TREAT algorithm discussed by Miranker [63] is similar in most respects to the RETE algorithm, except TREAT uses an additional source of information called *conflict set support*, which has been shown through quantitative experimentation to yield a more efficient matching algorithm.

production rule representations have been abstracted below.

```
(P ON-PHYS-OBJECT
  {(GOAL !STATUS ACTIVE !TYPE ON !OBJECT-NAME <O>)    <GOAL>}
  {(PHYS-OBJECT !NAME <O> !AT <P> !ON FLOOR)         <OBJECT>}
  {(MONKEY !AT <P> !HOLDS NIL !ON <> <O>)            <MONKEY>}
-->
   (FORMAT T "~2%Climb onto ~A~%" <O>)
   (MODIFY <MONKEY> !ON <O>)
   (MODIFY <GOAL> !STATUS SATISFIED))

(P ON-PHYS-OBJECT-AT-MONKEY
   (GOAL !STATUS ACTIVE !TYPE ON !OBJECT-NAME <O1>)
   (PHYS-OBJECT !NAME <O1> !AT <P> !ON FLOOR)
   (MONKEY !AT <> <P>)
-->
   (ASSERT* (GOAL !STATUS ACTIVE !TYPE AT !OBJECT-NAME NIL !TO <P>)))
```

As their names would suggest, both production rules belong to the same logical grouping. The particular group of rules they belong to is responsible for achieving goals which require the monkey to get on some physical object in its environment when various conditions exist (i.e. for particular world states). Notice how the structural similarities exhibited by the english statements have been preserved by the production rule encoding.

The intra-element and inter-element feature tests extracted from the condition parts of ON-PHYS-OBJECT and ON-PHYS-OBJECT-AT-MONKEY are summarized in Tables 3.4 and 3.5 respectively. Focusing for the moment on their intra-element feature tests, it is readily apparent that condition elements C0 and C1 of both rules specify the exact same list of pattern requirements. In addition, while the feature tests specified by their respective C2 condition elements are not identical, they do share the stipulation that slot 1 of any satisfactory wme must contain the symbol MONKEY. These pattern requirements are combined together, along with those from any other production rules, into a pattern network. The construction of the pattern network is accomplished in such a manner as to maximize the number of shared feature tests. For instance, the feature tests necessary for condition element C0 of both rules can be combined into a single series of tests with the provision that any wme satisfying the tests be distributed to the corresponding C0 condition of each rule. A pattern network constructed from the intra-element feature tests of ON-PHYS-OBJECT and ON-PHYS-OBJECT-AT-MONKEY is illustrated in Figure 3.4.

The structure of the pattern network shown in Figure 3.4 can be seen to be that of a rooted tree. Each path from its root node toward a leaf node represents one distinct series of intra-element feature tests. For example, the feature tests along the path of nodes labeled

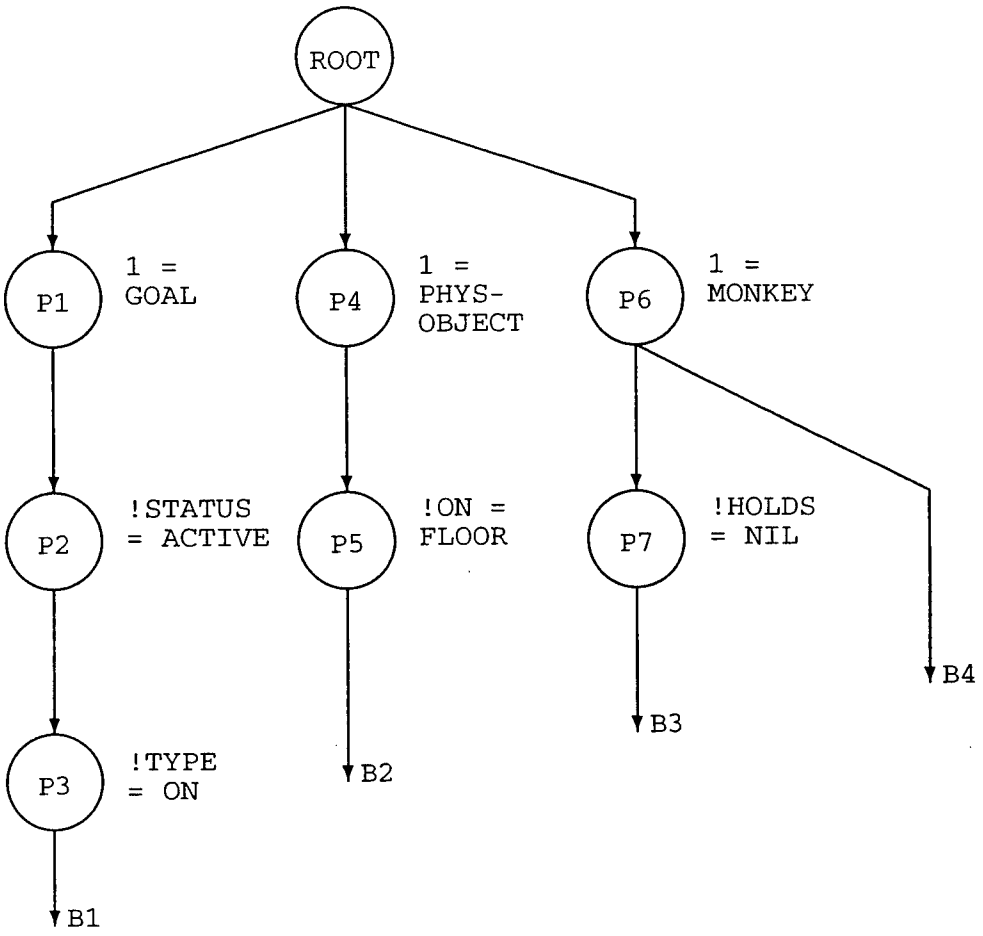


Figure 3.4: Example Pattern Network.

Intra-Element Feature Tests	
C0:	⟨(= 1 GOAL) (= !STATUS ACTIVE) (= !TYPE ON)⟩
C1:	⟨(= 1 PHYS-OBJECT) (= !ON FLOOR)⟩
C2:	⟨(= 1 MONKEY) (= !HOLDS NIL)⟩
Inter-Element Feature Tests	
C0 – C1:	⟨(= (C0 !OBJECT-NAME) (C1 !NAME))⟩
C0, C1 – C2:	⟨(≠ (C0 !OBJECT-NAME) (C2 !ON)) (= (C1 !AT) (C2 !AT))⟩

Table 3.4: Feature Tests for ON-PHYS-OBJECT.

Intra-Element Feature Tests	
C0:	⟨(= 1 GOAL) (= !STATUS ACTIVE) (= !TYPE ON)⟩
C1:	⟨(= 1 PHYS-OBJECT) (= !ON FLOOR)⟩
C2:	⟨(= 1 MONKEY)⟩
Inter-Element Feature Tests	
C0 – C1:	⟨(= (C0 !OBJECT-NAME) (C1 !NAME))⟩
C0, C1 – C2:	⟨(≠ (C1 !AT) (C2 !AT))⟩

Table 3.5: Feature Tests for ON-PHYS-OBJECT-AT-MONKEY.

P1, P2, and P3 represent the sequence of intra-element feature tests corresponding to condition element C0 as listed in Tables 3.4 and 3.5. Given a single working memory element, the production system interpreter performs a type of depth-first traversal of the entire pattern network. Along each path the interpreter draws upon the information contained in the encountered nodes to carry out a series of feature tests on the wme. Whenever one of these tests is successful, the interpreter continues its traversal to the next sequential node. However, if the wme fails any single feature test, the interpreter will abandon its traversal of the particular branch and will continue by visiting the next available node. In the event a wme passes all the feature tests specified by any condition element, the interpreter makes a copy of the wme and *deposits* it into an associated storage area, or *bucket*. Thus, the interpreter utilizes the structure of a pattern network to effect a kind of filter for partitioning working memory elements into groups possessing an assortment of intra-element features. It should be noted that a solitary wme may have suitable features entitling it to be deposited in any number of distinct buckets. Table 3.6 summarizes the association between the buckets (labeled B1–B4) in Figure 3.4 and the condition elements of production rules ON-PHYS-OBJECT (denoted by an 'X') and ON-PHYS-OBJECT-AT-MONKEY (denoted by a 'Y').

Thus far, the role of the production system interpreter has been described as one of traversing the pattern network in order to determine in which buckets a single working memory element should be *deposited*. Intuitively, this process could easily be extended to

	Condition Element		
Bucket	C0	C1	C2
B1	XY		
B2		XY	
B3			X
B4			Y

Table 3.6: Association Between Condition Elements and Buckets.

include the entire contents of working memory. The result would be a set of buckets, each containing a copy of the wmes which satisfied its specific pattern requirements. Since these pattern requirements were specified by the condition elements of individual production rules, the enumeration of rule instances can be viewed as a matter of correlating the contents of buckets associated with a particular rule according to its inter-element requirements. The data structure utilized by the RETE algorithm to facilitate such correlations is the join network. The join network consists of various *join* nodes which are used to *merge* paths extending from the buckets of the pattern network. Each join node has two inputs called the *left input* and *right input*, and a data storage area associated with each one called the *left memory* and *right memory*. In order to correlate the contents of n buckets (i.e. merge n paths of the pattern network into a single path), $n-1$ join nodes connected in a cascade type of arrangement are required. For example, to merge the paths leading from three nodes of the pattern network labeled A, B, and C would require two join nodes which will be referred to as J1 and J2. The first join-node, J1, can be used to merge paths A and B into a new path called AB, while join-node J2 is used to merge paths AB and C into the desired path ABC.

For matters of convenience, wmes were previously thought of as being *deposited* into various *buckets* according to the attributes they possess. In fact, when the interpreter reaches a node of the pattern network which completes a series of intra-element feature tests, rather than depositing a copy of the particular wme into a *bucket*, the interpreter places a copy into the corresponding memories of every join node connected to it. Consider the structure of the join network generated for the production rules ON-PHYS-OBJECT and ON-PHYS-OBJECT-AT-MONKEY shown in Figure 3.5. The left and right inputs of join node J1 now take the place of *buckets* B1 and B2 respectively as depicted in the pattern network of Figure 3.4. Exactly which nodes in the pattern network are to be merged is implicitly specified by the structure of each production rule LHS. When *compiling* a production rule, the last node traversed, [14] and/or inserted into the pattern network when integrating the intra-element feature tests of each condition element, is *marked* as a

[14] The intra-element features common to a number of condition elements may only require those nodes which have been previously inserted into the pattern network.

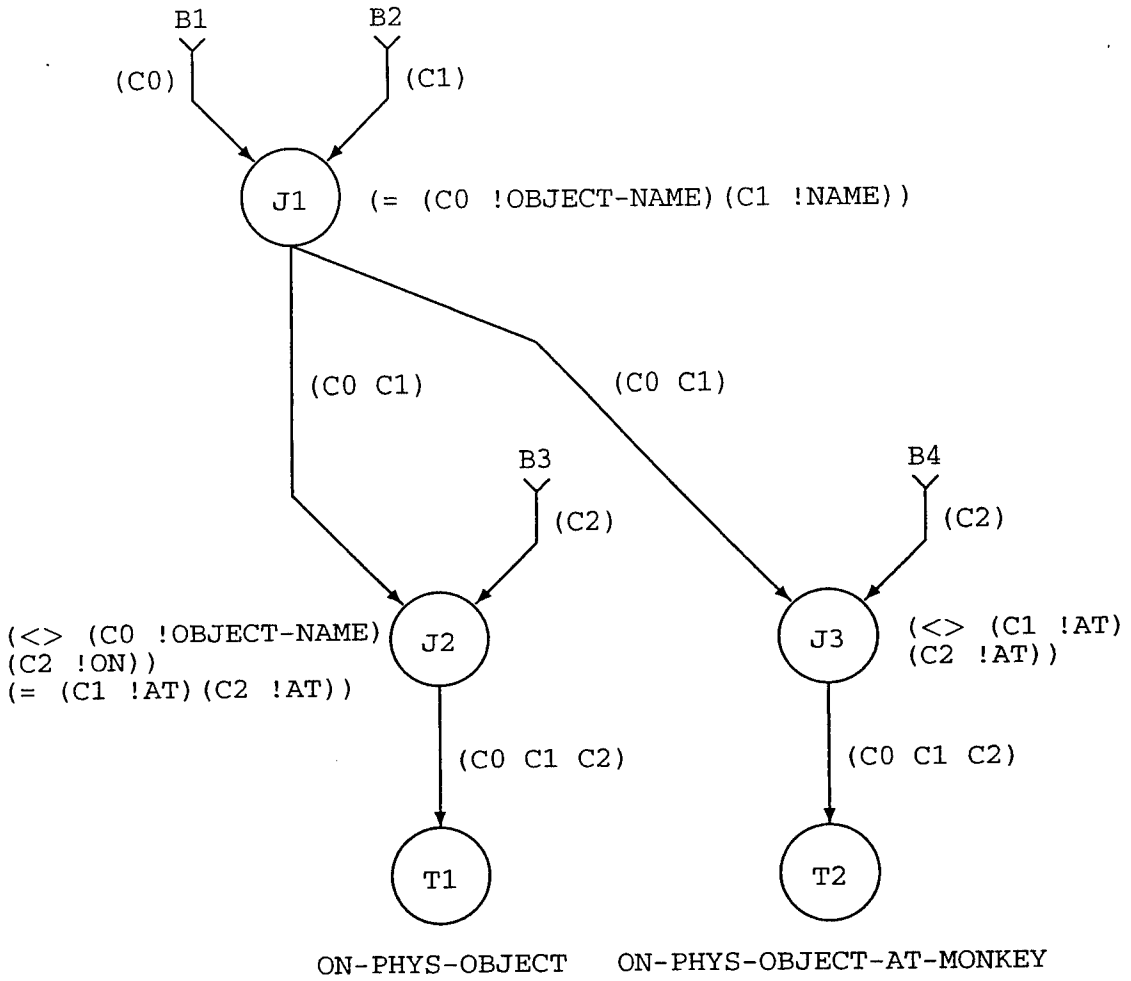


Figure 3.5: Example Join Network.

point of reference for modifying the join network. Since only the wmes which reach these marked nodes could possibly satisfy the production rule's respective condition elements, connections must be established to an appropriate cascade of join nodes which will perform the required inter-element feature tests. For example, the three nodes in the pattern network corresponding to condition elements C0, C1, and C2 of Table 3.5 are P3, P5, and P6 respectively, which are then merged together by join nodes J1 and J3 as shown in Figure 3.5. As with the pattern network, the join network can be made more efficient when the results of inter-element tests common to a number of production rules are combined into a "single" comparison and then distributed accordingly (e.g. the correlated output of node J1 is shared by nodes J2 and J3 in Figure 3.5).

All that remains to complete the discussion of the RETE pattern matching algorithm is to introduce the procedures by which join nodes are used to correlate data in their memories

in order to produce a series of changes in the conflict set. The basic data structure used for computing this correlation is called a token which represents an intermediate pattern matching result in the form of a partial match. Join nodes accept these tokens as input and produce *extended* tokens as output when a match for their inter-element feature tests is found. The general form of a token is the following:

$$(action\ wme_0\ wme_1\ wme_2 \ldots)$$

The first component of a token, the *action* parameter, is one of the LISP keywords :ASSERT or :RETRACT which is used to indicate to the nodes of the join network how the token is to be processed. The remaining parameters represent a partial match as an ordered sequence of working memory elements, such that, each wme_n matches condition element Cn of some production rule. This correspondence between token wmes and condition element matching is denoted in Figure 3.5 as a parenthesized list of condition element labels appearing alongside each join node connection. For example, the designation (C0 C1), located near the connection from the output of node J1 to the left input of node J2, indicates that all tokens traversing this path contain two wmes which match the corresponding C0 and C1 condition element of the rule ON-PHYS-OBJECT. By observing that such designations increase in size by one condition element label at each level of a particular join node cascade, a better understanding can be obtained of how partial matches are progressively generated. Since join nodes are cascaded by connecting the output of one node to the left input of another, all tokens received by the right input of any join node have descended directly from the pattern network. The information contained in these tokens can be seen to represent a single wme which has the pattern features specified by a particular condition element. Likewise, a token which propagates through the pattern network to the left input of a join node also matches a particular condition element, specifically condition element C0. A join node extends these tokens by examining the contents of other tokens it has received from its right input. If any matches are found (i.e. the inter-element feature tests are satisfied), the tokens received by the left input are extended by appending the wme contained in the tokens received from the right input to the end of the token received by the left input. Thus, the first join node always outputs tokens of the form (C0 C1). This process is continued inductively until a token representing a complete match is generated.

As an example, consider a portion of the join network corresponding to the production rule ON-PHYS-OBJECT duplicated in Figure 3.6, and the contents of working memory listed in Table 3.7. In the figure, the tables on either side of the join nodes represent the contents of their respective memories, and each memory entry is represented as a list of wme (fact) identifiers (e.g. F6 represents "fact" number 6) which were extracted from the received tokens. The example to be considered is one in which a new fact, F16, has just been added to the working memory. The state of the join network prior to this action is indicated by the unmarked memory entries. Those memory entries which are marked with an asterisk, were added as a consequence of the interpreter's processing of the new fact.

ID	Working Memory Element
F2	(PHYS-OBJECT !NAME LADDER !ON FLOOR !AT X3-Y7)
F5	(PHYS-OBJECT !NAME BIGBOX !ON FLOOR !AT X1-Y3)
F7	(PHYS-OBJECT !NAME LADDER !ON FLOOR !AT X5-Y5)
F10	(MONKEY !AT X5-Y5 !ON FLOOR)
F14	(GOAL !STATUS ACTIVE !TYPE ON !OBJECT-NAME BIGBOX)
F16	(GOAL !STATUS ACTIVE !TYPE ON !OBJECT-NAME LADDER)

Table 3.7: Example Working Memory.

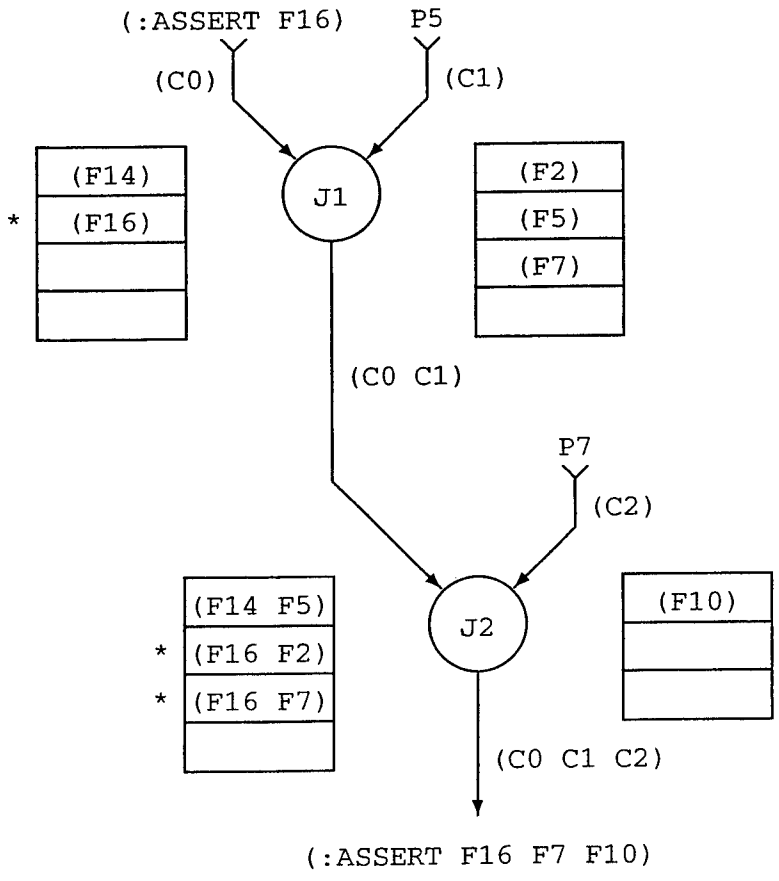


Figure 3.6: A Pattern Matching Example.

Before a new fact can be added to or deleted from working memory, an appropriate token must be created which contains the fact. For this example, the token (:ASSERT F16) is created for adding fact F16 to working memory and given to the match network interpreter. The interpreter then traverses the pattern network with this token until it performs the intra-element feature test associated with node P3 (refer to Figure 3.4). The token is then passed to the left input of join node J1. An action component of :ASSERT indicates the join node should store the contents of the token (i.e. a list of the token's wmes) in its respective memory. Likewise, an action component of :RETRACT indicates the join node should remove the token's contents from memory. For this example, join node J1 adds the wme list (F16) to its left memory. Having performed the appropriate memory storage operation, a join node next iterates over the contents of its *opposite* memory with the new information to determine which entries satisfy its inter-element feature tests. For join node J1, it will iterate over the contents of its right memory (i.e. wmes corresponding to condition element C1) using the new information (F16) for C0. Referring back to Table 3.4, the inter-element feature tests programmed into node J1 are:

$$\langle(= \text{ (C0 !OBJECT-NAME) (C1 !NAME)})\rangle.$$

While performing these tests, node J1 finds that only entries (F2) and (F7) are acceptable. Entry (F5) fails because its NAME attribute is BIGBOX and not LADDER. Join node J1 then uses these two entries from its right memory to generate the following two extended tokens:

(:ASSERT F16 F2)
(:ASSERT F16 F7).

These new tokens are then output from node J1 and distributed to the left input of join node J2. Again, the :ASSERT action parameter instructs node J2 to add these wme lists to its left memory, and perform the matching operation against the contents of its right memory. Referring again to Table 3.4, the inter-element feature tests performed by node J2 are the following:

$$\langle(\neq \text{ (C0 !OBJECT-NAME) (C2 !ON)) } (= \text{ (C1 !AT) (C2 !AT)})\rangle.$$

Comparing the contents of the first token against the right memory entry (F10) fails because the AT attribute of F2 is not equal to the AT attribute of F10 (i.e. the monkey is not by the particular ladder), and therefore cannot be extended. However, the contents of the second token does satisfy the inter-element feature tests of J2 and is therefore extended and output from J2 as:

(:ASSERT F16 F7 F10).

The contents of this token constitute an instance of the rule ON-PHYS-OBJECT and are passed to the *terminal node* T1 (Refer to Figure 3.5). Terminal nodes are used to convert received tokens into a corresponding production rule instance and add or delete them from the conflict set according to the action parameter of the token. An action parameter of :ASSERT indicates the generated rule instance should be added to the conflict set, while a parameter of :RETRACT indicates the rule instance should be removed from the conflict set, if it is present.

3.4.4 Conflict Resolution and Rule Execution

Conflict Resolution is a process by which one of the production rule instances contained in the conflict set is selected for execution. In TESS, the conflict set consists of an ordered list of instances stored in a data structure called an *agenda,* and the selected strategy for conflict resolution is used to determine how the entries in the agenda are to be ordered. TESS provides two conflict resolution strategies called SALIENCE and LEX, either of which can be selected by the user for ordering the agenda. To select the desired conflict resolution strategy, the user executes a function call of the following form:

(SET-CONFLICT-RESOLUTION-STRATEGY :*Strategy*).

Ideally, the selection of a particular strategy should be the first declaration of any production system. However, this can be accomplished anytime before a working memory element is asserted or deleted without error. If no such declaration is made, the SALIENCE strategy is selected by default. In both strategies, instances are mapped onto the set of integers which are used as keys for determining where in the agenda the instances should be inserted. An agenda can be thought of as a *priority queue* where the value of an instance's insertion key is its priority with the smallest integer key present being of the highest priority. The agenda is maintained as a list of instances in which those with the highest priority are located nearest the front of the list, and the first element of the list is always occupied by the next instance to be fired. When a new instance is generated, the keys for entries already in the agenda are compared one after the other to the new key. When an entry is found which has a key *greater than* the new key, the new instance is inserted into the agenda *before* that entry. In this way, a group of rule instances which all have the same value key will be inserted into the agenda in a manner which produces a First-In/First-Out (FIFO) type of rule execution. For example, consider the portion of an agenda shown below:

	Agenda	
Priority	Entry	Production Rule Instance
-13	0	(RULE01 F34 F0 F29)
0	1	(RULE07 F58 F23)
5	2	(RULE13 F0 F9 F10 F57 F13)
8	3	(RULE04 F47 F39 F1)
22	4	⋮

After inserting two new instances, first the instance (RULE07 F59 F23) and then instance (RULE07 F60 F23), both of which happen to produce a key of zero, the agenda now becomes;

	Agenda	
Priority	Entry	Production Rule Instance
-13	0	(RULE01 F34 F0 F29)
0	1	(RULE07 F58 F23)
0	2	(RULE07 F59 F23)
0	3	(RULE07 F60 F23)
5	4	(RULE13 F0 F9 F10 F57 F13)
8	5	(RULE04 F47 F39 F1)
22	6	⋮

Of the two strategies, LEX and SALIENCE, the SALIENCE strategy is the easiest to understand since it represents an explicit form of control. With this strategy, the mapping of production rule instances onto the set of integers according to their relative priority is declared using the optional argument to the function P. With no salience value specified, all instances of the particular rule will map to the integer 0 by default. For the purposes of understanding the operation of a production system, specifying saliences other than the default should be used cautiously and sparingly. When it does become necessary to explicitly control the order in which production rules are fired, the reasons for doing so and their effects on the system should be clearly documented.

Recall that each working memory element is assigned its own unique time tag (i.e. a positive integer incremented for each wme) at the time it is created. These time tags are utilized by the LEX conflict resolution strategy to impose a kind of lexicographical ordering on production rule instances stored in the agenda. The time tags of each wme which make up an instance are considered as a single key formed by a list of integers sorted in descending order. For example, the instance (RULE17 F20 F51 F32 F11) is taken to be the key (51 32 20 11). When comparing two keys, the dominance of one key over the other is established based upon which key contains the most *recent* time tags (i.e. the largest set of integers). Since a key is composed of a sorted list of integers, this determination is very similar to

a lexicographic ordering. Thus, this strategy is called LEX. As an example, consider the following list of keys:

1. (51 32 20)
2. (39 26 22 18 14)
3. (64 48 30)
4. (51 32 20 11).

Of the four keys, the one with the largest (and most recent) first element is the third key. Since there are no other keys with the same first element, the third key is dominant over all the others. The next highest first element is 51 and is present in both the first and fourth keys. In order to determine which of these two is dominant, their second elements must be compared. However, their second elements are also equal, as are their third elements; thereby making it necessary to compare their fourth elements. Because the first key does not have a fourth element, key number four is determined to be more dominant because it is more specific. Carrying this process through to completion yields the following ordered set of keys;

1. (64 48 30)
2. (51 32 20 11)
3. (51 32 20)
4. (39 26 22 18 14).

In the case where a number of keys are found to be equal, the keys are ordered randomly with respect to one another. With regards to the ordering of instances within the agenda, all instances under the LEX conflict resolution strategy are considered to have a priority of zero. Thus, unlike the SALIENCE strategy which orders instances chronologically with respect to a number of priorities, the LEX strategy orders instances lexicographically with respect to a single priority.

After a production rule instance has been selected for execution (i.e. the instance occupying the first element of the agenda), the rule name is removed from the instance leaving a list of matching working memory elements. Next, the compiled function corresponding to the rule's RHS is retrieved from a hash table using the rule name as a search key. Once the compiled function has been retrieved, it is called using the list of matching wmes as an argument and the *recognize-act* cycle begins another pass.

3.5 The Distributed Network Architecture

A cooperative problem solving agent incorporating the TESS system has been designed to execute in the SIMULACT distributed simulation environment. It consists of an actor process, configured to perform as a continuously running expert system, and a dedicated mail server for processing communications with other problem solving agents. Figure 3.7 shows a block diagram for this cooperative agent architecture. The structures labeled *IN-QUEUE* and *OUT-QUEUE* are buffers for holding incoming and outgoing mail messages respectively. Periodically, the mail server will check the agent's mailbox for any received messages. If there are messages, they are first organized by their time of arrival, then according to their relative priority and placed in the *IN-QUEUE*. These mail messages are then processed one after another by selecting an appropriate *handle* according to the type of message and who it is from. A handle is a function which has been specially designed to process a certain kind of message. Depending on the nature and content of a particular message, the corresponding handle function may expedite the message without ever involving the actor process. For example, a received message which was incorrectly formed or not understood may prompt a handle function to immediately dispatch a response by placing an error message in the *OUT-QUEUE*. However, generally, a handle function may place information into either the Interface Buffer for use by the actor process and/or into the *OUT-QUEUE*.

With this particular agent architecture, the information put into the Interface Buffer by the handle functions processing incoming mail messages are actually tokens for direct interpretation of the actor's pattern matching network. For received messages from similar agents, little is required of a handle function but to place the communicated tokens into the Interface Buffer. However, when processing a message from other kinds of agents, a handle function may have to put considerable effort toward translating the information from its native form into an acceptable token for a TESS system. Similarly, outgoing mail messages generated by an agent's expert system must have a way of transforming its basic unit of information (i.e. working memory elements) into suitable forms for other agents. For this purpose, a set of fact translation functions are provided which essentially perform the reverse procedures of their respective handle functions. The results of such translations are then placed into the *OUT- QUEUE* and periodically dispatched to the appropriate agents by the mail server.

As is the case with most production system interpreters, TESS employs a *closed world* assumption as an implicit means of ascertaining when an expert system has concluded its processing and should be terminated. A closed world assumption presumes that nothing exists beyond the extent of an expert system's own innate knowledge and stored data. This assumption is commonly used when matching wmes against production rules containing negated condition elements. If no wme matches, the rule may be instantiated (i.e. it is assumed that no wmes exist beyond the contents of working memory). Similarly, if there

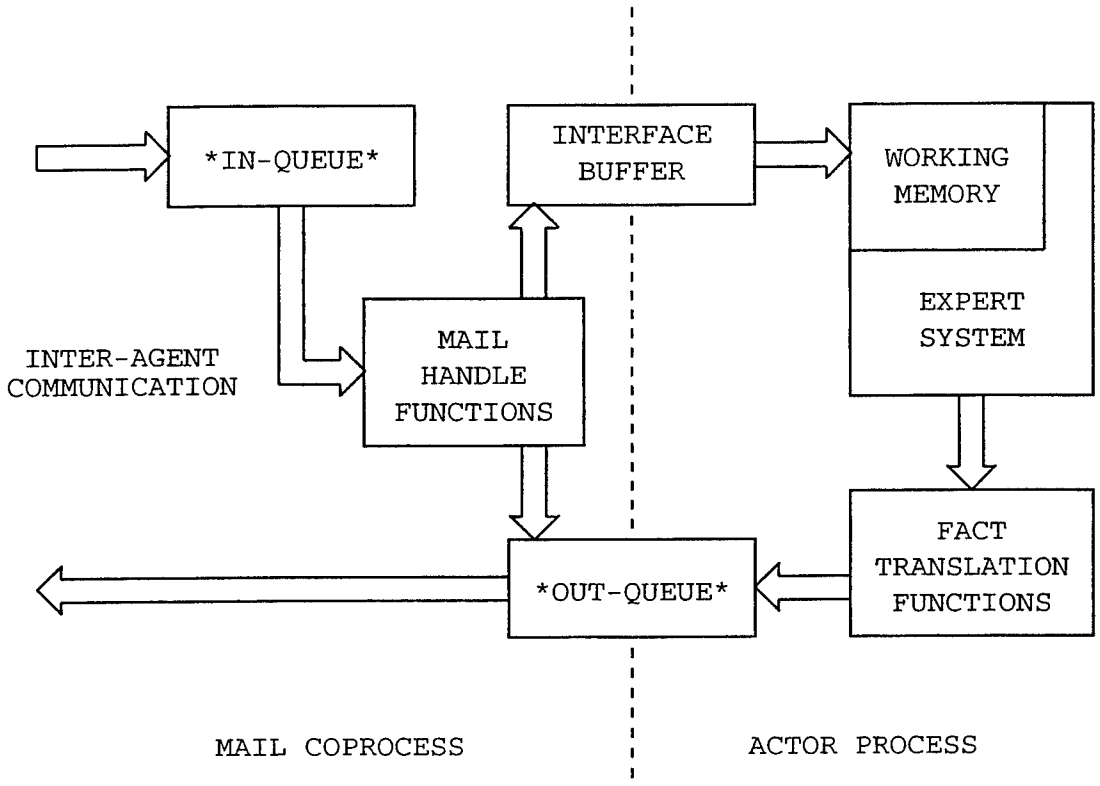


Figure 3.7: Cooperative Problem Solving Agent Architecture.

are no rule instances on the agenda given the current contents of working memory, the closed world assumption precludes any future instantiations and is therefore used as a criteria for terminating the production system. On the other hand, in a distributed environment such an assumption may prompt the interpreter to halt an expert system prematurely. For instance, if an agent executing an expert system sends a message to another agent and expects to receive a reply, a situation could arise where the agent's agenda becomes empty before a reply is received. In such a case, the closed world assumption would cause the agent's expert system to terminate before it was actually finished.

Complications of this kind can be overcome by treating the environment of an agent's expert system as a kind of *locally* closed world in which the existence of other knowledge sources beyond its own extents are periodically acknowledged. Pattern matching and program termination in a locally closed world are performed in the same manner as before, but with the utilization of a "snap-shot" of the working memory. The problem of premature termination of an expert system is circumvented by placing an agent's call to its interpreter (usually executed by its script function), within the body of an infinite loop. In this way, when the expert system terminates as a result of the closed world assumption, it can be restarted from its *previous* state when new information from other agents is received. A snap-shot of working memory allows an agent to ensure its contents will not change unexpectedly during an interpreter cycle. Any proposed changes to working memory received by external knowledge sources are held in the Interface Buffer until they can be properly incorporated into a succeeding snap-shot of the working memory. A *hook* programmed into the interpreter of TESS allows the recognize-act cycle to include other processing functions. This hook is used by expert system agents to check the Interface Buffer for changes to working memory by other external agents. When there are tokens present, they are interpreted one after another into the pattern matching network. Once this has been done, the interpreter proceeds to its next cycle.

Cooperative agents of the type described have been paired with DARES distributed theorem proving agents to produce an effective distributed problem solving system. Each pairing can be thought of as one node of a distributed network of nodes as shown in Figure 3.8. In this architecture, each TESS agent is expert at formulating distributed reasoning problems based upon its local interpretation of the problem domain. This formulation is then passed to each agent's respective DARES counterpart. Having accomplished this, the DARES agents are then signaled to begin their problem solving activities. Once some conclusion has been reached by the DARES system, the data derived by each DARES agent is transferred to their respective TESS agent.

3.5.1 Distributed Domain Extensions of TESS

Modern software engineering guidelines dictate that low-level implementation details should be keep transparent to a user in order to provide a uniform set of system interfaces. In

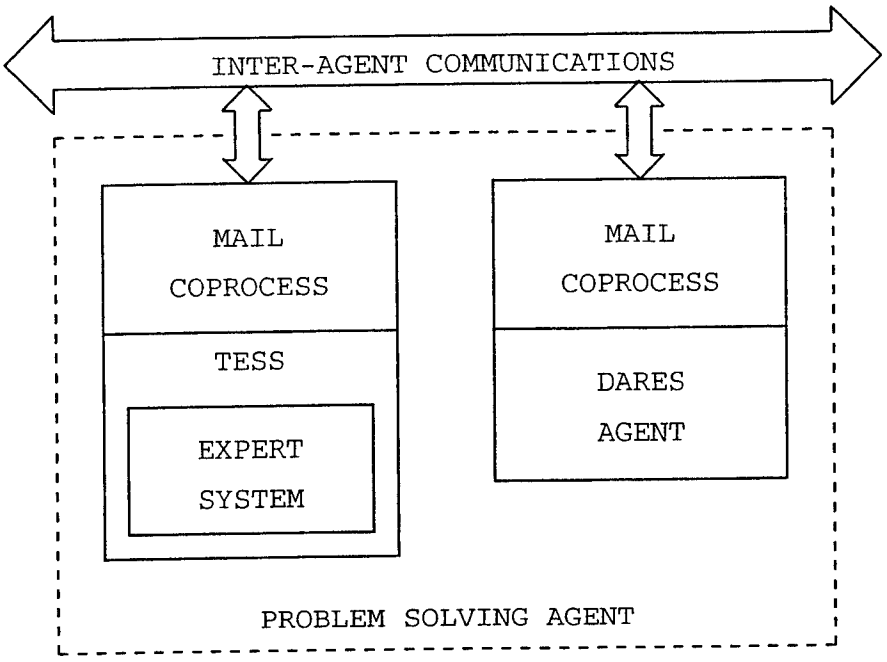


Figure 3.8: Architecture of a Distributed Problem Solving Node.

keeping with this policy, two extensions to the TESS system have been especially installed for distributed domains under the SIMULACT simulation environment. These extensions allow an agent to assert and retract various facts from *external* working memories. In other words, for agents designed around the TESS support package, any agent can communicate with and in fact control if need be, another agent simply by placing *foreign* facts into its working memory. Two functions closely resembling the use of ASSERT* and RETRACT have been provided for this purpose. These functions are appropriately called ASSERT-FAR and RETRACT-FAR. They have the following calling structure:

(ASSERT-FAR *stagename* (*attribute-value list*0)
(*attribute-value list*1)
⋮
(*attribute-value listn*))

(RETRACT-FAR *stagename* (*attribute-value list*0)
(*attribute-value list*1)
⋮
(*attribute-value listn*)).

The only visible difference between these functions and their "near" counterparts is

the additional *stagename* argument which is evaluated. As a matter of convention, the stagenames given to various distributed agents have two parts, a *genus* and an *identifier*, separated by a hyphen. The genus describes what kind of distributed agent an actor is and the identifier specifies which agent in the genus a particular actor is. For example, the stagename TP-C could be used to specify agent C of a distributed theorem proving (TP) system. The form of a stagename is significant to these functions and how the attribute-value lists are treated. When the genus of the agent specified by the stagename argument is of the same type[15] as the agent firing the rule, the attribute-value lists are simply sent to the specified agent using a memo. That agent's mail server will select the appropriate handle function for processing them. In the case of an ASSERT-FAR, the lists are used by the receiving agent to generate new wmes which are then made into tokens with an :ASSERT action parameter. On the next cycle of the agent's interpreter, these new tokens will be incorporated into its match network. However, in the case of a RETRACT-FAR function call, all the elements of the receiving agent's working memory which match the specified lists of attributes and values, are used to make tokens with an action parameter of :RETRACT. Thus, all those matching wmes will be removed from the agent's working memory on its next interpreter cycle.

When communicating with agents of another genus, a special translation function (refer to Section 3.5) must have been previously defined for transforming the information contained in the attribute-value list into a representation suitable for the particular agent. In this way, agents built around the TESS expert system shell can interface with many disparate types of system agents.

3.6 Application to the Defense Communication System

The Defense Communication System (DCS) is a world-wide military communication network that has been upgraded by replacing its analog equipment with comparable digital technology. With the introduction of this new digital equipment there is the opportunity for implementing a more efficient form of network control. Ordinarily, when a problem arises in the network, it is handled by the coordinated efforts of a group of technical control personnel. When the incidence of network failures becomes increasingly more frequent, or when there are many simultaneous faults caused by some catastrophic failure, the judgment and decision-making abilities of the technical controllers can be impaired by high levels of stress. This problem can be alleviated by automating many of the more tedious decision making processes currently performed by the technical controllers. By presenting the technical controllers with accurate preprocessed information, they are better able to make the *correct* network control decisions.

[15] For agents within a given genus, a shorthand can be used where only the agent's identifying symbol needs to be specified.

To automate some of the more difficult decision making processes of a technical controller, the automated system must have the capability to coordinate its efforts in a similar fashion to that of the technical controllers. Since a large portion of the tasks performed by a technical controller are recognition tasks, a distributed situation recognizer can be used to coordinate the efforts of an automated system. An example of how this can be done in the DCS domain will be presented in section 3.6.4. Before one can thoroughly appreciate such a detailed example however, a good understanding of communication networks is necessary as well as some of the more specific details associated with the DCS and its implementation. The following sections should be easily understood by a reader who is not familiar with communication networks and may be useful as a refresher for more knowledgeable readers.

3.6.1 Communication System Concepts

A communications network is sufficiently complex for it to be worthwhile spending some time to briefly discuss the concepts underlying its implementation. Without understanding these few basic concepts, it would be extremely difficult, if not impossible, to fully understand and appreciate a detailed example. This is especially true of an example based on the physical implementation of a communications network. The most basic concept is that of a *channel.* A communications channel can be thought of as a pathway that allows information to be exchanged between two or more parties. To avoid loosing any part of a communication, the method of exchanging information through the channel must be understood and adhered to by all parties involved. There are generally two such modes of communication known as *half-duplex* and *full-duplex.*

In the half-duplex mode of communication, the exchange of information in the channel is in one direction only. At any particular time only one party is designated as the transmitter, while the other is designated as a receiver. If both parties have the facilities to either transmit or receive, then two-way communications are possible in the half-duplex mode by alternately designating each party as either a transmitter or a receiver. In the case where one, or both, parties do not have both transmit and receive capabilities, the half-duplex mode of communication permits either *send-only* or *receive-only* types of information exchanges.

The full-duplex mode of communication is possible when the channel between two communicating parties supports simultaneous bi-directional exchanges of information. Each party has the ability to transmit and receive information and is allowed to do so at the same time. When using electronic transmission media, such as a cable or microwave link, a full-duplex channel is necessarily composed of two half-duplex channels. One channel is always designated as send-only while the other is always designated as receive-only, with respect to a single party. This is the most versatile type of communications channel and the type generally used in most communication systems.

One important aspect of a communications channel is the rate at which information can

be exchanged. A communications channel has physical limitations that determine an upper bound on the amount of information it can carry in a given period of time. This is known as the *bandwidth* of a channel. The bandwidth of a communications channel imposes both economical and architectural constraints on the design of a communications system. For example, if a particular application requires two 500 Kbit/sec channels to be constructed between the same locations and the only channel bandwidths available are 500 Kbit/sec at $750/mile and 1.5 Mbit/sec at $1000/mile, two options need to be considered. The first is to use two parallel 500 Kbit/sec channels at a cost of $1500/mile, and the second is to use a single 1.5 Mbit/sec channel to carry the information of both channels. Clearly the second option would minimize the cost of implementation, and in addition, would provide another 500 Kbit/sec of bandwidth for other applications.

Although greatly simplified, the example described above does illustrate the usefulness of grouping several channels together and transmitting their information on one channel. In communication terms, a channel used in this way is known as a *trunk channel*, or simply a *trunk*. It should be pointed out however, that a trunk channel must have *at least* the bandwidth of its constituent channels combined. To make this point clear, consider a channel that transmits information at a rate of 250 Kbits in one second and another channel that transmits information at a rate of 300 Kbits in one second. In parallel, these two channels are capable of transmitting a total of 550 Kbits each second. It should be obvious that a 500 Kbit/sec channel is not capable of transmitting the same amount of information in one second as the two channels in parallel. Therefore, to carry an equivalent amount of information as the two channels in parallel, a trunk channel with at least a bandwidth of 550 Kbit/sec is necessary.

Figure 3.9 depicts a simple communications trunk between two black boxes labeled X and Y. For the sake of clarity, they will be referred to simply as X and Y. All that is known about the black boxes is that each one has two dedicated sections, one section is a transmitter and the other is a receiver. The transmitter section combines two channels into a trunk channel and the receiver section separates a trunk channel into its constituent subchannels. By connecting the output of the transmitter section of X to the input of the receiver section of Y, a send-only half-duplex channel is constructed from X to Y. A similar channel is constructed from Y to X completing the full-duplex trunk channel labeled T. The two input channels X1 and X2 of X are combined and transmitted through trunk T where they become the respective output channels of Y. Channels Y1 and Y2 are transmitted in a similar fashion from Y to X. Two full-duplex channels can be transmitted over trunk T by making associations between pairs of channels. Channel X1 can be grouped with channel Y1 and denoted as the full-duplex channel X1Y1, and likewise, channel X2 can be grouped with channel Y2 and denoted as the full-duplex channel X2Y2.

As an example, consider the problem of connecting a pair of telephones together from X to Y. By arbitrarily choosing input channel X1 of X and connecting it to the pickup coil of the X telephone and connecting the X1 output channel of Y to the voice coil of the

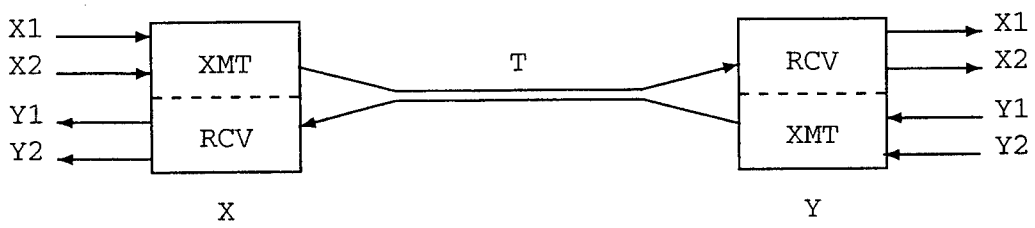


Figure 3.9: A two-channel full-duplex communications trunk.

Y telephone, a person using the Y telephone could listen to a person speaking into the X telephone. To complete the hookup, a similar connection can be made using channel Y1 in the opposite direction. The result is a complete telephone *circuit* X1Y1 that provides simultaneous speaking and listening capabilities for each party. Since T is a two-channel full-duplex trunk, a second, and totally separate, telephone circuit could be established using the remaining channels X2 and Y2 in a similar fashion.

It should not be surprising that trunk channels can be combined with other trunk channels to make even larger trunks. This process of combining trunks can be continued to form a trunk hierarchy. Figure 3.10 illustrates a simple two-level full-duplex trunk hierarchy using four black boxes, identical in function to those of the previous example, labeled W, X, Y, and Z. It is assumed that trunk T1 carries two full-duplex channels W1Z1 and W2Z2 from W to Z by first combining with channel X2Y2 to form trunk T2. Whenever one channel combines with another channel to form a trunk, the constituent channels are said to *ride* the trunk. For example, in Figure 3.10 channels W1Z1 and W2Z2 both ride trunk T1 which in turn rides trunk T2. Channel X2Y2 does not necessarily have to be a trunk with the same structure as trunk T1 to be able to ride trunk T2. Depending on the local architecture, channel X2Y2 may be a point of termination. Channels can be terminated in many ways by devices such as a telephone as in the previous example, or a computer terminal. In general a channel can be terminated by any device that sources/sinks data at a rate which is compatible with the bandwidth of the channel.

In a communications network there are a number of *stations*, or *sites,* which are interconnected by high-speed trunk channels. In general, each site has several trunk connections to other sites. Typically, a trunk that enters a site may have some of its channels terminated, and possibly replaced with channels from other trunks and/or termination equipment, before being redirected to another site. In this manner, the required circuits for communication can be established by following a path over the necessary trunks from site to site until their destinations are reached. As may have been evident from Figure 3.10, the trunks and channels formed by this simple type of routing architecture, like those discussed thus far, terminate on similar black boxes at the same hierarchical level. To make this point

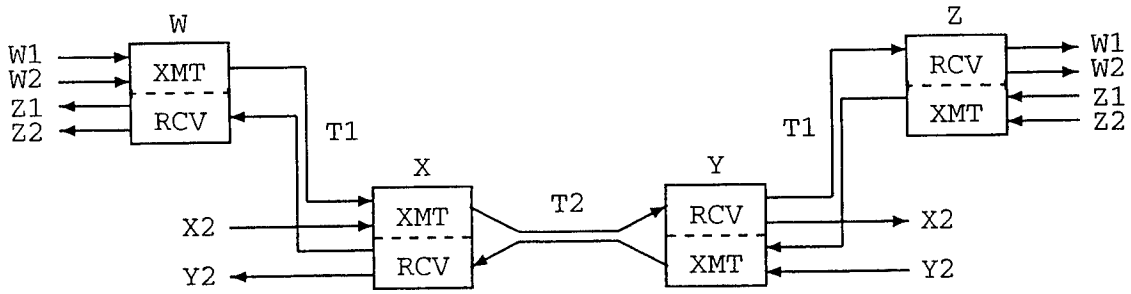


Figure 3.10: A two-level communications trunk hierarchy.

more clear, Figure 3.10 has been redrawn as Figure 3.11 using the more conventional single-line representation of full-duplex channels and with more emphasis placed on a network interpretation.

When trunk T2 is viewed as a high-speed channel linking the two sites labeled S1 and S2, the *mirror* quality exhibited by trunk hierarchies is readily apparent. The lowest level of a trunk hierarchy is considered to be the level at which the slowest channels are combined into trunks. From the previous discussion concerning the necessary *speed* of trunks over that of their constituent channels, it should be clear that W and Z combine the slowest channels in their respective sites. Black boxes W and Z are therefore considered to be the first level in the trunk hierarchy as shown in Figure 3.11. Likewise, while the functions performed by black boxes X and Y are the same as those performed by W and Z, it should be clear that X and Y must necessarily be capable of performing theses functions at least twice as fast as W and Z. Since black boxes X and Y combine the next highest speed channels, they are considered to be the second level in the trunk hierarchy.

This structural property of trunk hierarchies in a communications network has particular advantages for a technician troubleshooting a problem in the network. For instance, if W were indicating that it has a problem receiving data from trunk T1 and X appears to be functioning properly, the technician can eliminate possible causes by coordinating with another technician located at site S2. Together the technicians can verify the operation of the black boxes at every level in the hierarchy traversed by trunk T1 until the transmission problem is isolated. From the point of view of the technician located at site S1, W and X are considered to be *local* devices. Likewise, a black box at another site that terminates the same channel as a local device is referred to as that local device's *distant end*, or *d/e*, device. For example, black box Z is the distant end device of W as viewed from site S1.

Up to this point a number of concepts and terms relevant to the underlying physical implementation of the *transmission network* of a communications system have been discussed. Typically, a communication system is visualized as having several layers where the transmission network is considered the most fundamental layer. It is the layer where all

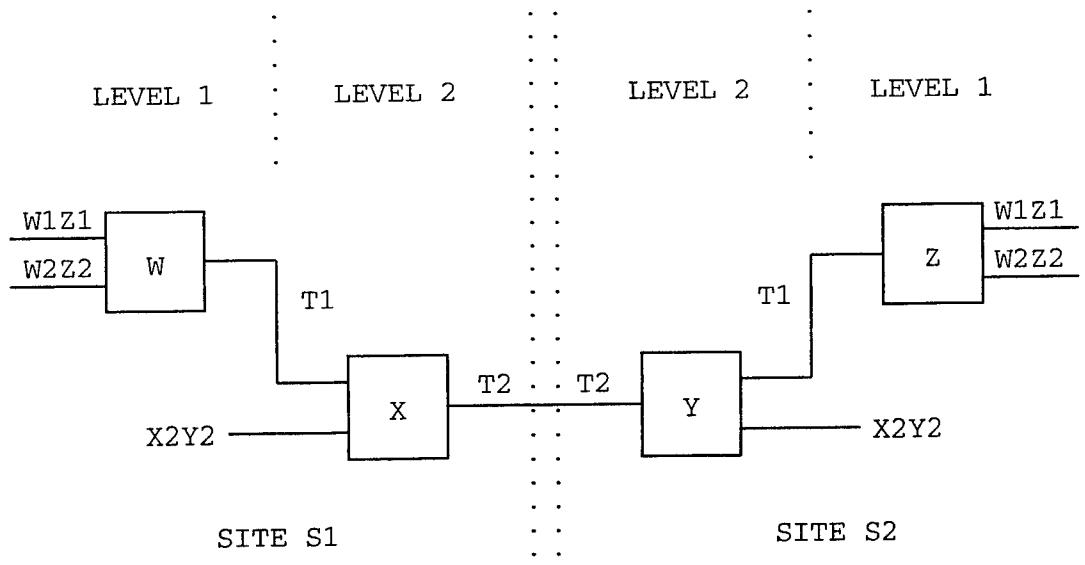


Figure 3.11: A two-site communications network.

signals are physically transferred from one site to another on a path to their destination. The next layer above the transmission network is called the *switch network*. The switch network is generally viewed as a network of gates that determines what path a call will take in getting from its origin to its destination. Once a path has been determined, the call is *switched* onto the appropriate channels and trunks of the transmission network . For the purposes of an example, consideration of a communications system will be limited to maintaining the integrity of the transmission network.

3.6.2 The DCS Transmission Network

In the DCS, a system known as Transmission Monitoring and Control (TRAMCON) is intended to oversee and maintain the operation of its transmission network. The TRAMCON system, although not yet fully implemented, will be responsible for detecting and isolating equipment failures that have a direct impact on the *performance* of the transmission network. In addition, after having verified an equipment failure, TRAMCON will be responsible for finding alternative routes for as many of the affected channels as possible according to their relative priority. Due to the natural geographic distribution of the transmission network and the relatively high cost of communications, a distributed problem solving approach to the TRAMCON problem has been adopted. In a communications network, like the DCS, there may be hundreds of individual sites making it impractical, and inefficient, to represent each site as a distributed processing *node*. Instead, the TRAMCON system groups the sites into disjoint subsets called *subregions*. Each subregion has one site designated as the controller of all the other sites in the subregion. The site designated as the

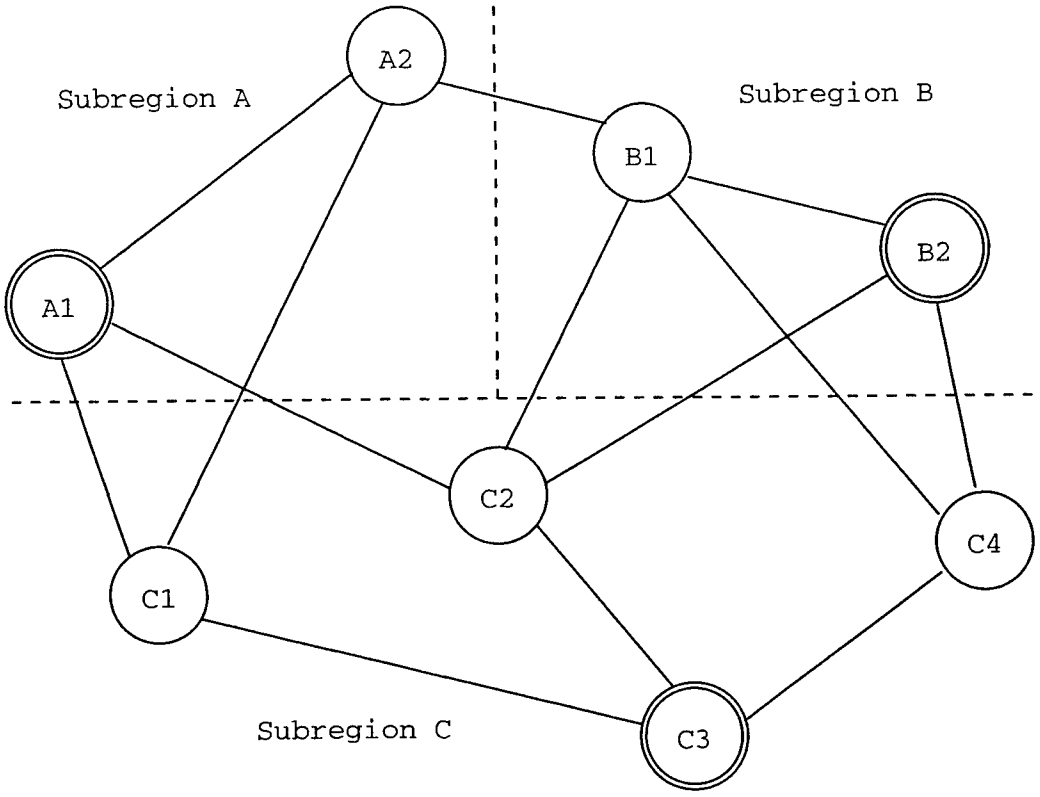


Figure 3.12: A simple network divided into subregions.

subregion controller is referred to as the Subregion Control Facility (SRCF) and represents a processing *node,* in the distributed problem solving sense, of the TRAMCON system.

Figure 3.12 illustrates an example of how the sites of a small communications network may be grouped into three subregions labeled A, B, and C, as shown delineated by dashed lines. Each circle in the figure represents a site in the network and is identified by the label shown in the center of the circle. Those sites which are designated as the SRCF for their subregion are represented using a second larger circle concentric with the one used for an ordinary site. For example, sites A1, B2, and C3 in Figure 3.12 are designated as the SRCFs for subregions A, B, and C respectively. A labeling convention often used, and the one adopted in this report, is to assign labels to each of the sites in a network according to the subregion they are a member of. For instance, the sites which are a member of subregion C have been assigned the labels C1, C2, C3, and C4. In this way, the subregion a particular site belongs to can be identified simply by looking at the label it was assigned.

Every communications site in the DCS is supplied with a piece of equipment known as a DATALOK-10, or as simply a DATALOK. A DATALOK is used to monitor the operation of other equipment in the site which are considered to be *important* to the functioning of

the transmission network. It does this by temporarily storing the status and alarm signals generated by each piece of equipment until this information is requested sometime in the future. The SRCF of each subregion communicates with the DATALOKs at all the other sites in the subregion using a dedicated service channel. A SRCF continually polls the DATALOKs of each site in its subregion in a round-robin fashion. When a DATALOK is polled by a SRCF, it responds by transmitting back to the SRCF its current set of status and alarm signals, reported by the equipment it monitors, since the last polling period. In this way, each SRCF site in the network incrementally updates its view of the problems being experienced by *important* pieces of equipment throughout the entire subregion.

3.6.2.1 TRAMCON Node Architecture

The local problem-solving architecture of each SRCF node in the TRAMCON system is logically distributed into three functional units (i.e. problem-solving agents) known as Fault Isolation (FI), Performance Assessment (PA), and Service Restoral (SR) as shown in Figure 3.13. Another agent, the Knowledge-Base Manager (KBM), is used to ensure the views of each of the other agents remains consistent with the *known* state of the subregion. To accomplish this task, the KBM is given detailed apriori knowledge about the equipment present at each site in the subregion, their local connectivity, and their connectivity with other sites [15]. The KBM is also given knowledge to reflect the information gathered from the DATALOKs during each polling period. Whenever one of the agents requires knowledge from the KBM, all that is necessary is for the agent to make an appropriate request and the KBM will forward a reply sometime shortly thereafter. However, for the KBM to be an effective manager, the knowledge *derived* by each agent must also be *registered* with the KBM. This is especially true of the knowledge derived by one agent that may have an impact on the processing being performed by another. As an aid in maintaining a consistency among the *beliefs* registered by each agent, the KBM employs the MATMS[41] truth maintenance system.

The role of each Performance Assessment agent is a central one for initiating problem-solving activity within the TRAMCON system. As the alarm and status information is gathered from each site in the subregion, the PA agent attempts to *quickly* interpret this information and assess its impact on the performance of the transmission network. In most cases, an assessment can not be made solely on the information available to a single subregion, some distributed problem-solving effort by a group of PA agents is necessary. During this problem-solving activity, each PA agent registers its beliefs concerning the operational status of equipment and associated trunks, channels, and circuits with its respective KBM. When the PA agents have completed their assessment, each one notifies its respective FI and SR agents as required.

As mentioned above Performance Assessment makes a quick, or first-cut, estimate concerning the cause of the problem implied by the reported status and alarm signals. Because

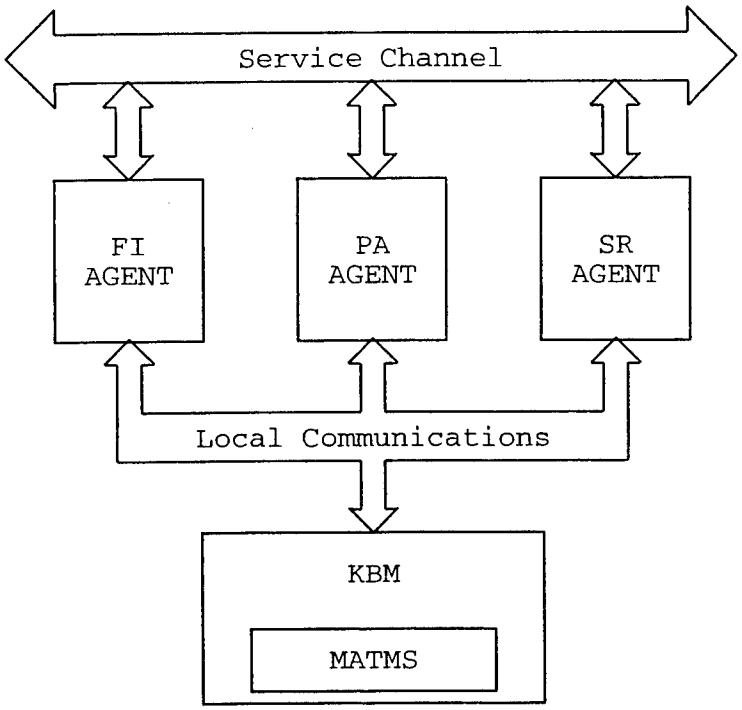


Figure 3.13: The local architecture of each TRAMCON node.

of this, the likelihood of Performance Assessment making an error is considerable. It is for this reason the PA agents notify Fault Isolation of equipment it has decided are not operating properly. The role of Fault Isolation is to methodically and meticulously interrogate the operation of each piece of equipment named by PA. If FI determines that a piece of equipment is in fact not operating correctly, then FI and PA beliefs about the equipment are consistent and nothing needs to be done with the knowledge-base. On the other hand, if FI determines that a piece of equipment is functioning properly, the beliefs of FI and PA on the status of this equipment are contradictory and must be resolved. Fortunately, based on the purpose of each agent the MATMS can, in good conscience, adopt a policy of always believing an FI agent over a PA agent. In this case, the PA agent would have to be notified that its belief was incorrect so that it could incorporate this more believable knowledge.

The Service Restoral agents rely heavily on the beliefs asserted by the agents of Performance Assessment. Once Performance Assessment has notified Service Restoral of particular circuit outages, the SR agents work together trying to find alternative routes for those circuits. To do this, each SR agent must request, from the KBM, which channels are available to carry the circuits. The response from the KBM will not include those channels considered by its respective PA agent to be *down,* which otherwise may have been useful. After receiving this information, the SR agent must register with the KBM that it believes all these channels to be *up* and in good working order. Always registering such beliefs

ensures that Service Restoral will only consider rerouting circuits using reliable channels.

The need for an SR agent to register its beliefs can be made clear by considering a potential problem involving contradictory beliefs between PA and SR agents similar in nature to the one between FI and PA agents. Suppose Performance Assessment has been notified that a belief it previously asserted has been updated by Fault Isolation and it incorporates this new belief. The consequence of this action would cause Performance Assessment to re-evaluate its data and update its beliefs in the KBM. This situation has the potential of changing the status of a channel from *up* to *down* which is being used by Service Restoral to generate a rerouting plan. However, since Service Restoral *knows* nothing about trunks, channels, circuits, etc., the beliefs of Performance Assessment are clearly superior to those of Service Restoral. This problem is easily resolved by allowing the MATMS to adopt a policy of always believing a PA agent over an SR agent. In this way, SR agents affected by the change in status would be notified by their KBM to no longer include that channel in their plans.

3.6.2.2 TRAMCON Status and Alarm Signals

Equipment that generate status signals usually implement them as digital pulses which indicate that some operational *transition* has just been made. To capture this data for transmission at the next polling period, a DATALOK must latch, or *lock in,* the signal. As an example of the type of information a status signal represents, consider a piece of equipment that transmits and receives a trunk channel. If for some reason the equipment suspects a malfunction of its on-line receiver, it may issue a status signal which essentially says *"Attention, I'm using my backup receiver now."* to indicate that it has switched over to its redundant receiver. Alarm signals are similar to status signals but are more directly related to a problem. An alarm signal indicates that a piece of equipment has detected an internal fault or there is something wrong with the data being transmitted or received. A typical example of what two such alarm signals may indicate could be interpreted as *"Hey, my power supply is failing!"*, or *"Hey, I'm not receiving any data!"* respectively. Alarm signals are implemented using digital logic voltage levels to represent their active and inactive alarm conditions and do not require latching by a DATALOK. Under normal circumstances an alarm signal maintains its inactive voltage level, and throughout the time interval a problem continues to exist, the alarm signal is held at its active voltage level.

In the previous section it was mentioned that DATALOKs only report the status and alarm signals of *important* pieces of equipment back to their SRCF, but there was no further mention of what it meant to be important. The TRAMCON system considers any piece of equipment used to implement the highest three levels of a trunk hierarchy to be important. TRAMCON neglects all other levels of the trunk hierarchy and refers to the three highest levels as level one, level two, and level three. The equipment at each of these levels are also ranked by their relative importance to the network making level one equipment the

least important and level three equipment the most important. It should be clear from previous discussions that these three trunk levels consist of the highest speed channels in the network and hence carry the most communications traffic. This strategy allows TRAMCON to maintain the parts of the transmission system which have the greatest impact on the performance of the network by reducing the amount of necessary information.

Although the exact nature of the alarm signals that may be generated by important equipment has not yet been discussed, it should not be surprising that a single failure could lead to a string of alarms along each path affected by the failure. In a substantially complex communication network, hundreds of alarms could potentially be generated from a single failure depending on its exact nature and level of importance. To further reduce the amount of information TRAMCON must deal with, the alarm signals generated by equipment at each level are arranged in order of their level of importance. For example, the alarms *recognized* by TRAMCON are more numerous and meaningful at level three than those at level two. The same is true of alarms generated at level two compared to those generated at level one. TRAMCON accomplishes this by progressively 'ORing' more of the equipment generated alarm signals into a single alarm as the level of their relative importance decreases. In this way, when any one of the equipment generated alarm signals is active, a single representative alarm signal is activated and recognized by TRAMCON.

Thus far three strategies used by the TRAMCON system have been discussed to reduce the complexity of the problem of maintaining the transmission network. To summarize these strategies, they are; 1) use a SRCF to represent an entire subregion, 2) emphasize only the three most important levels of the trunk hierarchy, and 3) generalize the equipment alarm signals based on the equipment's relative importance. While these strategies are very effective at reducing the complexity of the distributed problem, they are not without their associated costs. Consider the polling mechanism employed by the subregion strategy, it introduces uncertainty about the accuracy in a SRCF's knowledge of the subregion. This condition arises because a site relays local alarm conditions back to the SRCF as they are when the site was polled. As the SRCF polls the next site in its cycle, it is possible the local alarm conditions reported by the previous site are no longer valid. Generally speaking, a worst-case problem for this strategy would cause each site to completely change its alarm conditions immediately after it has been polled, making it impossible to solve the problem. Fortunately, the nature of a transmission network causes the majority of alarm conditions generated by a site to persist from one polling cycle to the next, making it more likely that a solution to the problem will be found. For the purposes of this report, it will be assumed that any network problem causes a set of alarms to be generated which remain constant throughout the problem solving process.

The second and third strategies summarized in the above paragraph limit the amount of usable information the TRAMCON system can apply to a problem. By considering only the most important levels of the trunk hierarchy, TRAMCON does not have access to local information within a site which may help verify a network problem. Instead, it may be

necessary for the TRAMCON system to consider a large number of related alarms in order to make the best verification possible. This introduces an uncertainty as to the problem itself and makes solutions all that much more difficult to find.

3.6.3 Transmission Network Equipment

To understand the details involved in monitoring and controlling a transmission network, like that of the DCS, a good deal of knowledge about the actual equipment used for its implementation is needed. The purpose and operation of the DATALOK-10 has previously been discussed, but thus far no mention has been made of the equipment actually used to implement a trunk hierarchy. Before going into any detailed descriptions of this equipment, it would be prudent to first understand some of the more basic concepts related to data transmission and synchronization which are common to each type of equipment. Afterward, a description of each piece of equipment used to implement TRAMCON trunk levels one, two, and three will be presented. Finally, one last piece of equipment used for cross-connecting channels between trunks will be described and will complete the knowledge necessary for an example in this domain.

In the communications domain, equipment used to perform functions similar to those of the black boxes mentioned earlier, are known simply as *multiplexers*. Although they are referred to as multiplexers, it is generally understood they perform both multiplexing and demultiplexing functions simultaneously in a manner similar to that modeled by the black boxes. The hardware used to implement a multiplexer is designed to work with either digital or analog data. An analog multiplexer senses the voltage at each of its input channels and replicates them in a *time-slice* of the trunk channel producing an output which is a segmented voltage waveform. A digital multiplexer is simpler in design and capable of performing its function much faster than its analog counterpart. Each input channel to a digital multiplexer can be thought of as a stream of data represented as a sequence of binary digits. The binary digits of each input channel are gated into a *frame* and transmitted serially on a trunk channel as a stream of bits, or *bit-stream*. Due to their simplicity of design and higher operating rate, the DCS is currently upgrading the analog multiplexers now in use to digital multiplexers for its transmission network, and will be the only type considered here.

A digital trunk channel can therefore be viewed as a stream of frames transmitted from the output of a multiplexer to the input of its distant-end device. Every frame is constructed from a number of bits, representing an equal amount of data from each of its constituent channels, together with an identifier to synchronize the multiplexed data. For example, consider the problem of transmitting four separate 14 Kbit/sec channels over a single 64 Kbit/sec trunk channel. Each of the four 14 Kbit/sec channels transmits seven bits of data (one ASCII code) in exactly 500 μs, while the 64 Kbit/sec trunk channel transmits 32 bits in the same period of time. In light of this, an obvious choice for the size of a frame

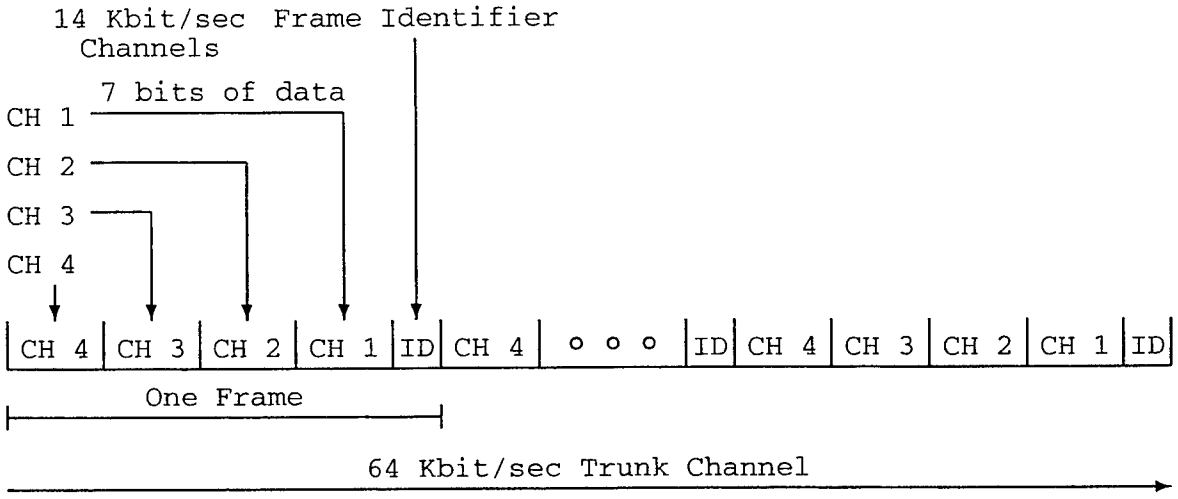


Figure 3.14: A four channel frame structure.

is to use 32 bits with seven bits allotted to each of the four channels in each frame. The remaining four bits are assigned to a special *frame identifier* code as illustrated in Figure 3.14. A frame identifier is a mutually agreed upon code for recognizing the beginning of each frame in a bit-stream. The transmitting device inserts the identifier into each frame and the distant-end device looks for the identifier in each frame to verify its synchronization. If the distant-end device were not synchronized (i.e. it is decoding a string of data offset by at least one bit), the data being demultiplexed from the trunk channel would be meaningless.

While the four bit frame identifier code used in the above example was convenient, typically they need to be much larger (e.g. 16 bits) to reduce the number of data strings within a bit-stream which could possibly be recognized as a frame. If there were a problem with the transmission of a trunk channel somewhere between the output of the multiplexer and its distant-end, or the channel were just opened, the distant-end device must somehow be able to resynchronize itself with the trunk channel. It does this by initiating a synchronization algorithm where the incoming bit-stream is first scanned for an occurrence of a bit pattern that matches that of the frame identifier code. Once found, the device *assumes* this pattern *frames* the multiplexed data. However, if the following string of bits, assumed to be the next frame, does not contain the proper identifier code, the search algorithm starts over. When two consecutive frame identifier matches are found, the device *assumes* it is now synchronized. Once a device declares itself to be synchronized, it keeps track of the number of consecutive times it fails to find the correct identifier within a frame. If this number becomes too large (e.g. over 123), the device *assumes* it has lost synchronization and initiates the resynchronization process again.

3.6.3.1 The Level 1 Multiplexer (MUX-98)

The AN/FCC–98(V) Multiplexer Set [83], known as a MUX-98, is a full-duplex device for simultaneously transmitting and receiving digital information over a communications trunk. In the transmission network, MUX-98s are used to implement TRAMCON level one trunk structures. As such, they are the most accommodating of the three types of multiplexers to be discussed. A MUX-98 has the capacity to multiplex a total of 24 voice frequency (VF) channels, or 18 data channels into a single trunk channel called a *digital group,* or *digroup trunk.* Other combinations of VF and data channels are also possible depending on the speed of the data and whether it is synchronous or asynchronous. Each channel to be multiplexed interfaces with the MUX-98 by using one of the available plug-in port modules.

A port is a *location* in the MUX-98 architecture where channel data are transmitted to the multiplexer module and received from the demultiplexer module as illustrated in Figure 3.15. Each communications channel to be multiplexed requires exclusive use of at least one of the 24 ports available in a MUX-98. Some applications may need a channel with a bandwidth broader than is provided by a single port location. For these applications a port module can be configured to utilize more than one port location, thereby providing a channel with additional bandwidth. As an example of the function performed by a typical port module, consider the problem of interfacing an ordinary four-wire telephone with a MUX-98. This can be done using a special voice-frequency (VF) channel module. A VF channel module converts an analog input signal into a pulse-code modulated (PCM) digital signal for transmission and converts a received PCM digital signal back into an analog output signal. A variety of other modules are also available for interfacing with such equipment as a low-speed multiplexer, teletype, computer terminal, DATALOK, etc.

In the 24-channel mode of operation, the MUX-98 uses a frame structure consisting of 193 bits. Each VF channel, or its data channel equivalent, contributes 8 bits of data to each frame for a total of 192 bits. The last remaining bit is reserved for a special frame identifier. The rate at which these frames must be transmitted and received is determined by the frequency of data used to digitally encode a voice frequency analog signal. For voice communications, a good quality telephone service can be provided by using only the frequency components of a voice in the range of 4 KHz. To digitally encode this signal into a PCM signal requires a sampling rate of at least twice that of the highest frequency, or 8000 data samples per second. Therefore a digroup trunk must transmit and receive data at a rate of 8000 frames per second which translates to a bandwidth of 1.544 Mbit/sec. When a MUX-98 is in this mode of operation, a digroup trunk is equivalent to a standard T-1[16] communications trunk. In the examples that follow, it will be assumed that each MUX-98 is configured for a 24-channel mode of operation and that digroup trunks are interchangeable with T-1 trunks. Figure 3.16 shows the schematic symbol for a MUX-98 that will be used

[16] In the communications domain T-1 is a standard unit of trunk capacity. When a trunk is described as being a T-1 trunk, this means the trunk is capable of carrying the equivalent of 24 (voice-frequency) channels.

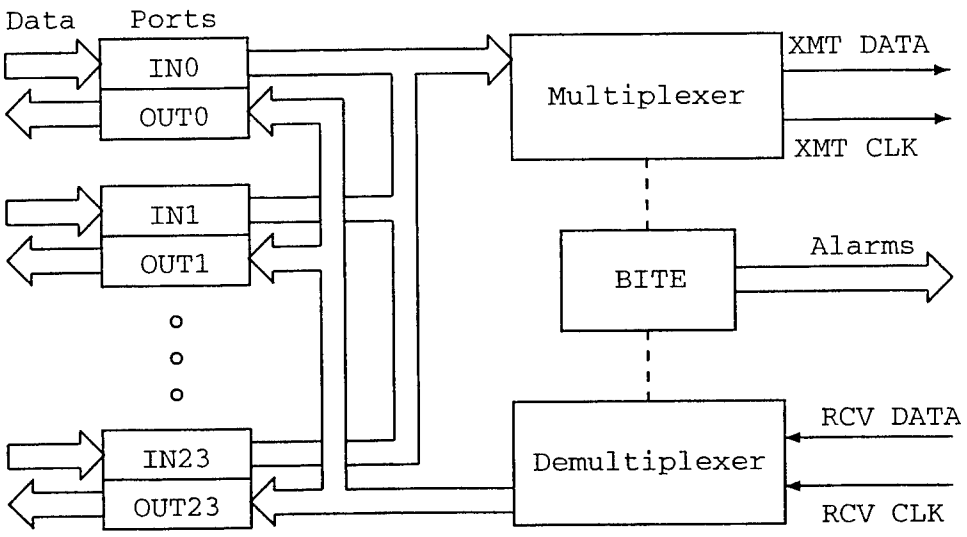


Figure 3.15: Functional block diagram for a MUX-98.

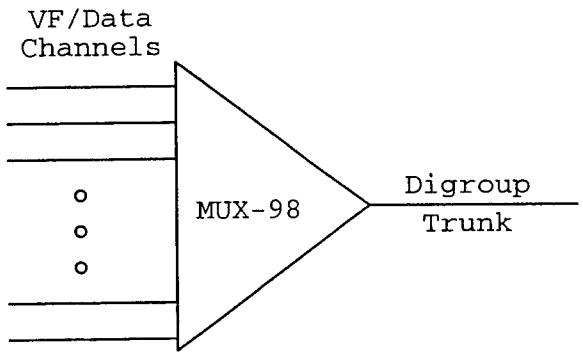


Figure 3.16: Network schematic symbol for a MUX-98.

in the example communications networks.

The MUX-98 has a Built-In Test Equipment (BITE) module which contains circuitry for monitoring the terminal characteristics of the multiplexer, demultiplexer, power supply, and port modules. By observing the data flowing into and out of each module, the BITE is able to determine whether or not the multiplexer set is functioning properly. Some of the problem conditions detected by the BITE module are; 1) loss of input or output data from a port module, 2) loss of synchronization by a demultiplexer module, 3) transmission problems with a multiplexer module, and 4) a loss of power from a power supply module. In the event any of these conditions have been detected, an appropriate alarm signal is generated by the BITE module and remains active for as long as the condition persists.

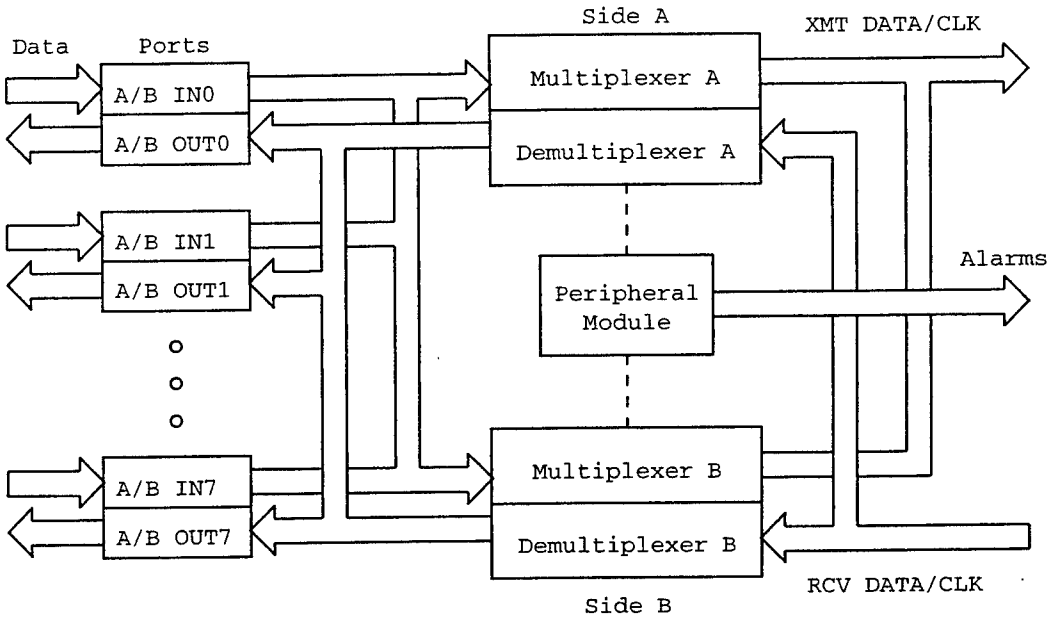


Figure 3.17: Functional block diagram for a MUX-99.

In the TRAMCON system all of these generated alarm signals are represented by a single general alarm which may be interpreted as *"Something is wrong with my digroup trunk!"* Due to the lack of information conveyed by such an alarm signal, some indication of what it may imply must be derived from other sources.

3.6.3.2 The Level 2 Multiplexer (MUX-99)

The AN/FCC–99(V) Multiplexer Set [84], known as a MUX-99, is very similar in function to that of the MUX-98 multiplexer. It is used to implement second level trunk structures in the transmission network. Unlike the MUX-98 however, the MUX-99 has two functionally redundant systems referred to as the A-side and B-side multiplexers. Both sides of the MUX-99 are identically configured to perform the same multiplexing function. At any point in time only one side actually performs the required multiplexing function and is designated as the *on-line* system, while the other side is designated as being the *off-line* system. In the event the on-line system develops a problem which would degrade its performance, the MUX-99 can *switch-over,* replacing the on-line system by the multiplexer that was off-line. A switch-over operation can be initiated in one of three ways; 1) manually by a local operator, 2) remotely with a command from a SRCF, or 3) automatically by the peripheral module. Figure 3.17 shows a functional block diagram of the MUX-99 multiplexer. The MUX-99 is designed to multiplex a total of 8 digroup trunks into a 12.352 Mbit/sec Mission Bit-Stream (MBS), also known as a *supergroup,* or *supergroup trunk.*

In the MUX-98 there was a BITE module for monitoring the operational status of the other modules and generating alarms when problems where detected. The MUX-99 contains a similar module, called the peripheral module, which performs the same type of monitoring functions as the BITE module, but in a more sophisticated manner. The peripheral module is able to take advantage of the dual architecture by continually performing a series of five tests on the two systems, to more accurately identify operational faults. The first test is used to establish the proper operation of the off-line system. By connecting the *output* of the off-line multiplexer to the *input* of the off-line demultiplexer and supplying generated data, the output of the demultiplexer can be compared with the data being supplied. If the two sets of data match, then the off-line system is determined to be functioning properly.

Using the known operating condition of the off-line system as a basis for comparison, the peripheral module can now accurately test the on-line multiplexer and demultiplexer modules. The second test is similar to the first one, but instead of using the output of the *off-line* multiplexer, it uses the output of the *on-line* multiplexer and makes another comparison using actual transmission data. If the data from the output of the off-line demultiplexer is the same as that being transmitted, then the on-line multiplexer is determined to be operating normally. In the third test a similar technique is used to verify the operation of the on-line demultiplexer. The peripheral module connects the input of the *off-line* demultiplexer to the input of the *on-line* demultiplexer so they are both receiving actual trunk data. A comparison of their outputs will determine whether the on-line demultiplexer is also operating normally.

The last two tests performed by the peripheral module are designed as a self-test to ensure that it is also functioning correctly. A repetition of the first test is performed to once again verify the operation of the off-line system. Having done this, the combined operation of the comparator and data generator is tested by *looping* the generated data through the off-line multiplexer and demultiplexer as in the first test. The data generated is modified with a 1 % bit error rate before being input to the multiplexer. By comparing the generated data and the data output from the demultiplexer, the comparator should find them identical *and* be able to detect the 1 % bit error rate. If it does not, the test fails and a peripheral fault alarm is generated. This cycle of internal testing is performed approximately 23 times each second and is quite reliable as long as a peripheral fault is not indicated.

The peripheral module of the MUX-99 detects all the same faults as did the BITE module of the MUX-98 with respect to either of the redundant systems. In addition, the peripheral module is capable of detecting 1) a loss of data being transmitted or received on the MBS, 2) how many frame errors have been found, and 3) the number of seconds synchronization has been lost. There are also several status signals generated that specify in what way a switch-over was initiated and which side of the MUX-99 is currently on-line. Figure 3.18 shows the schematic symbol that will be used to represent a MUX-99 in sample communication networks.

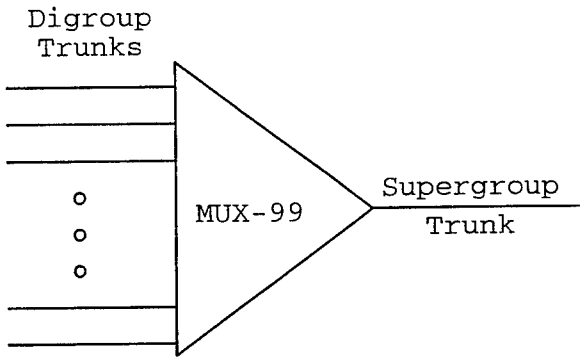


Figure 3.18: Network schematic symbol for a MUX-99.

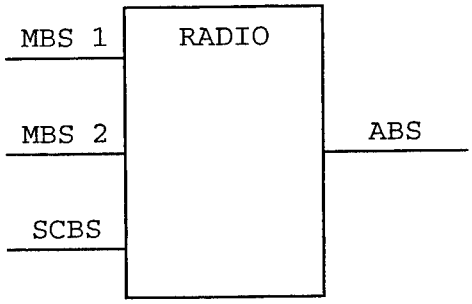


Figure 3.19: Network schematic symbol for a RADIO.

3.6.3.3 The Level 3 Multiplexer (RADIO)

The AN/FRC–171(V) Radio Set [85], known simply as a RADIO, is used to implement level three of the transmission network and is by far the most important and complex piece of equipment recognized by the TRAMCON system. A RADIO multiplexes two supergroup trunks and a single digroup trunk into a Aggregate Bit-Stream (ABS) for transmission over a microwave link to other communications sites in the network. The two supergroup trunks are referred to as MBS-1 and MBS-2, and the digroup trunk is what is known as the Service Channel Bit-Stream (SCBS). The SCBS is typically interfaced with a MUX-98, yielding 24 channels that can be used for such purposes as allowing a SRCF to communicate with a local DATALOK, interfacing with a TRAMCON terminal, etc. Figure 3.19 shows the schematic symbol that will be used to represent a RADIO in sample communications networks.

In Figure 3.20 one can see the multiplexing section (everything left of the transmitters and receivers) is very similar to a MUX-99. The peripheral module for this part of a RADIO performs much the same functions and generates the same alarm signals as the MUX-99. The RADIO is further complicated by the dual transmitter and receiver tacked onto the

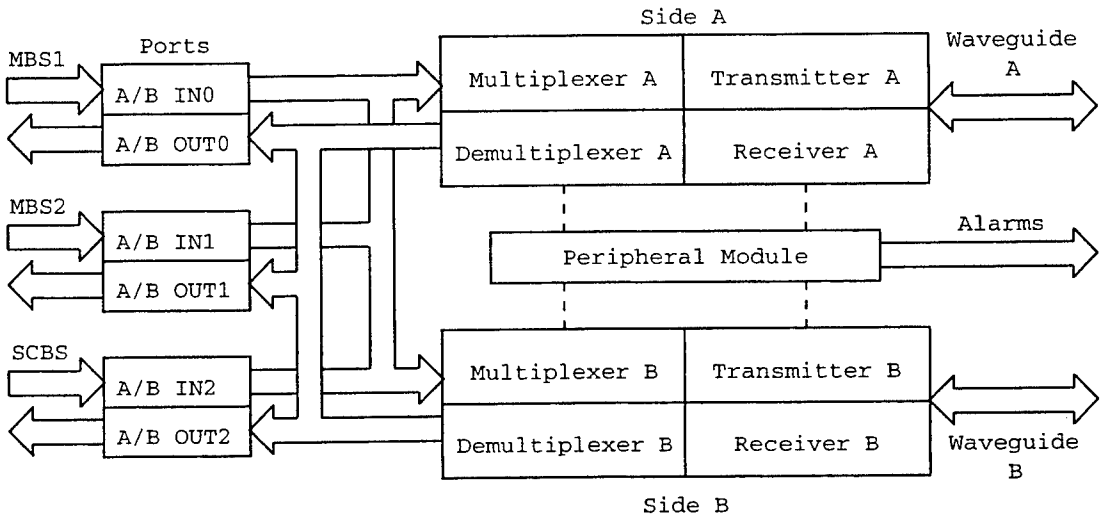


Figure 3.20: Functional block diagram for a RADIO.

right. Each transmitter *section* contains modules for scrambling the data, modulating the data onto a microwave carrier signal, and sending it through a waveguide and into a *dish* for transmission. Similarly, the receiver *section* contains modules for descrambling, demodulating, and receiving the microwave signal from the waveguide. Additional alarms are generated for conditions associated with the radio portion of this unit. The additional conditions detected are; 1) transmitter frequency drift, 2) transmitter power failure, 3) failure of a modulator or demodulator, 4) bad signal quality, and 5) receive signal loss.

3.6.3.4 The Digroup Trunk Channel Switch (DPAS)

When the DCS transmission network is fully upgraded to use digital equipment, each communications site will be equipped with a Digital Patch and Access System (DPAS). A DPAS[17] is a software controlled switch that functions as a digital *patch matrix* for interconnecting the channels of various trunks. The DPAS is designed to interface with all the T-1 data trunks in a site and provide arbitrary circuit switching capabilities between them. To better understand its function, Figure 3.21 illustrates how the channels of one trunk may be *patched into* the channels of another trunk using a DPAS. Channels A1 and A2 of trunk A are patched into channels B2 and B3 of trunk B respectively, and channel A3 of trunk A is patched into channel B1 of trunk B. In general, a DPAS can cross-connect the 24 channels of a connected T-1 trunk to the channels of any other connected trunks in an arbitrary way.

A DPAS is designed to be a *junction box* between the output of all MUX-98s and the

[17]Pronounced dee-pas.

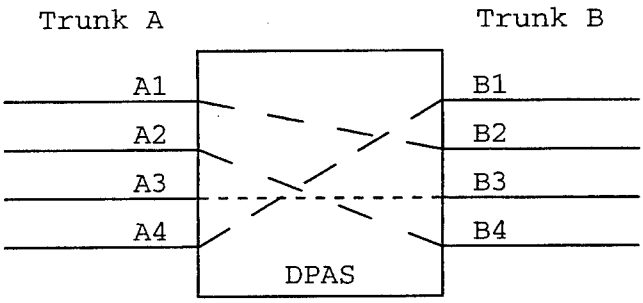


Figure 3.21: Channel patching function of a DPAS.

input of all MUX-99s. In this position, it allows the TRAMCON system to control and access every channel in a communications site. A DPAS can be used to reduce the amount of equipment required in a site by *packing* all the terminating channels onto as few as possible digroup trunks. In this way, only the minimum number of MUX-98s are used to terminate the channels rather than using a separate MUX-98 for every under utilized trunk used to carry them. Exactly how the channels switched by the DPAS are interconnected, is controlled by the TRAMCON system. The DPAS is used by TRAMCON to implement a circuit rerouting plan generated by the Service Restoral system. It also serves as Fault Isolation's interface to the individual channels of the transmission network. By providing access to these channels, FI agents are able to diagnose equipment failures by monitoring and performing tests on the individual circuits affected by a fault.

3.6.4 An Example

In this section a typical problem associated with the DCS transmission network will be discussed from the global perspective of a human expert. The objective here is to identify the expert knowledge that would be required in order to solve the problem. The architecture, equipment specifications, and communication paths of a particular network are described graphically using a program called GUS. GUS enables an individual to design an entire transmission network suitable for performing experiments and simulations. The output of GUS is used to supply the knowledge base for each KBM agent and also describes a set of script that describes how the status and operating conditions of individual network components is to change throughout the course of a simulation. In the lingo of GUS, these scripts are referred to as event scenarios.

For the purposes of this work, SIMULACT is being used to design and simulate the necessary facilities which would enable the TRAMCON system to be automated using a network of distributed problem solving agents. The DCS transmission networks to be used in these simulations are first described using a graphical design tool called GUS[40]. GUS

allows a network designer to precisely describe a DCS transmission network by graphically specifying various network objects and their interconnections. Among the objects to be described using GUS are sites (including SRCF designations), subregions, multiplexers, radios, trunks, microwave links, etc. When the network designer has completed the graphical description of a particular transmission network, the structural knowledge extracted from this description are partitioned into a distributed knowledge base. The partitions separate a network's structural knowledge along its subregion boundaries to form the knowledge bases located at each SRCF.

Recall that the TRAMCON system consists of three distinct distributed problem solving systems superimposed upon the same network of nodes, each of which correspond to an individual SRCF. These distributed systems are called Fault Isolation (FI), Performance Assessment (PA), and Service Restoral (SR). Each distributed problem solving node contains a single FI, PA, and SR agent which are modeled in the simulations as actors. Another agent, also modeled as an actor, called the Knowledge Base Manager (KBM) is also located at each problem solving node and is responsible for supplying and maintaining all network knowledge pertinent to its respective subregion. The knowledge base given to the KBMs at each problem solving node are exactly the partitioned structural knowledge bases provided by GUS which correspond to the particular subregion. Each agent obtains knowledge from its respective KBM agent using futures, and communicates with its own agents located at other processing nodes in a predetermined way using the available mail objects as provided by SIMULACT.

The event scenarios used to drive transmission network simulations are defined with the use of GUS in much the same way the network itself was defined. GUS has expert knowledge about the kinds of alarms and status signals recognized by TRAMCON for each piece of equipment in the network. Using this information, GUS allows an event (alarm) scenario to be defined by selectively choosing alarms and their time of activation for various pieces of equipment. When all the alarms for a particular event scenario have been defined, GUS organizes this information into a file ordered by alarm activation times. The sequence of alarms contained in this file are injected into a simulation by a ghost whose script function has been specially designed for this purpose. During each time interval in the simulation, the alarm ghost checks its list of alarm activations and if there are alarms whose activation time is equal to the current simulation time, the ghost dispatches memos to alert a particular SRCF of the alarm activation.

3.6.5 The Example Network

A small transmission network has been designed using GUS and will serve as a basis for discussing many of the intricate details associated with this application domain. Portions of this network spanning three subregions encompassing, for the most part, the scope of the example are shown in Figures 3.22 and 3.23. Due to the spatial limitations and a

desire to reduce the overall visual complexity of the network, all unnecessary connections have been truncated and given a label consisting of a site name in square brackets. These labels indicate the site at which the truncated connection's distant-end device is located. One of these truncated connections, microwave link L9, has been especially identified by an encircled X to denote the interconnection of Figures 3.22 and 3.23. The physical boundaries of each site are easily be identified by considering the location of RADIOs and MUX-98s as a sort of *perimeter* that discriminates the equipment of one site from another. In the network diagrams, each site has been given a label, consistent with the previously described naming convention, and is shown in bold-face located somewhere within the site's general vicinity.

The architecture of the transmission network depicted in Figures 3.22 and 3.23 is such that all digroup trunks within each site terminate on a DPAS. While this type of equipment configuration has the advantages of reducing the number of required MUX-98s and promoting higher levels of trunk utilization, it has the disadvantage of increasing network complexity. Each DPAS represents a point in the network where two channels from different digroup trunks may be arbitrarily interconnected. Since with this equipment architecture all digroup trunks eventually terminate on a DPAS and each digroup channel carries a single circuit, the paths of each circuit through the network can be described by specifying how these channels are switched at each DPAS along the way. For this purpose, a notation of the form [x:y] will be used to represent the channel y of trunk x. With this notation the interconnection of channel 1 of trunk T1 and channel 2 of trunk T2 can be represented by the pair ([T1:1], [T2:2]). The path of a circuit through the network can be represented by a sequence of these interconnections. For example, the path of circuit C10 from M08 to M32 in Figure 3.22 is described by the following sequence of connections:

$$C10: \quad \langle([T14:2],[T12:1])\ \ ([T12:1],[T39:3])\rangle$$

3.6.6 Expert Knowledge Required For a Solution

It is instructive to discuss how a group of human experts (i.e. the technical control personnel located at each SRCF) would approach a typical problem. In large communication systems, technical controllers are often confronted by problems which do not substantially affect the performance of the transmission network, however. These same problems often do require significant coordination efforts by the technical controllers to accurately identify, and subsequently restore, any lost network services. An example of a network problem with this characteristic is that of interpreting the pattern of alarms generated throughout a network which is the result of a single port module malfunction in a MUX-99. With regards to the network of Figures 3.22 and 3.23, the specific problem that will be considered here is the one in which the port module that interfaces trunk T45 with MUX-99 M31 in site A5 is defective.

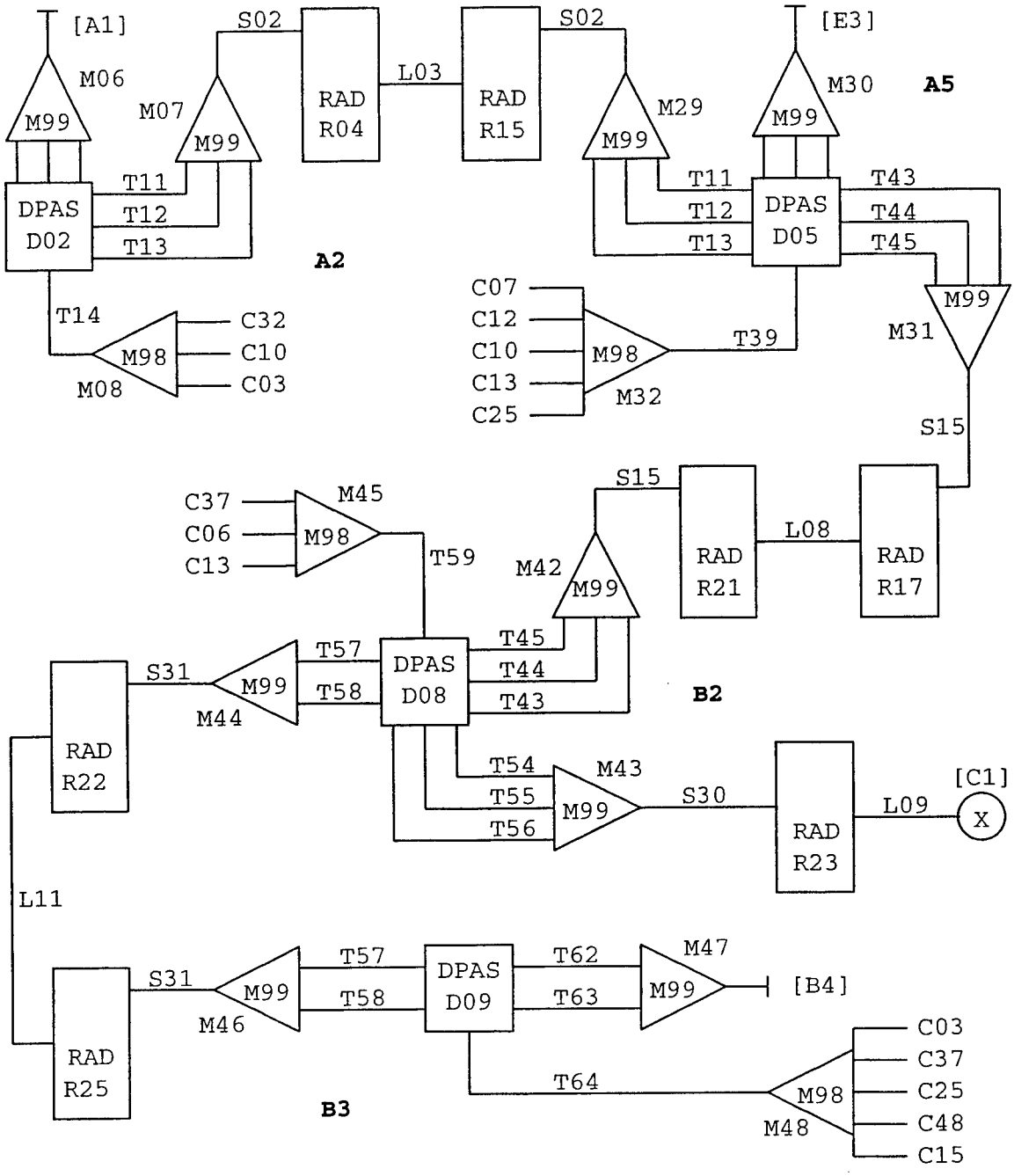


Figure 3.22: Example Communication Network (Part 1).

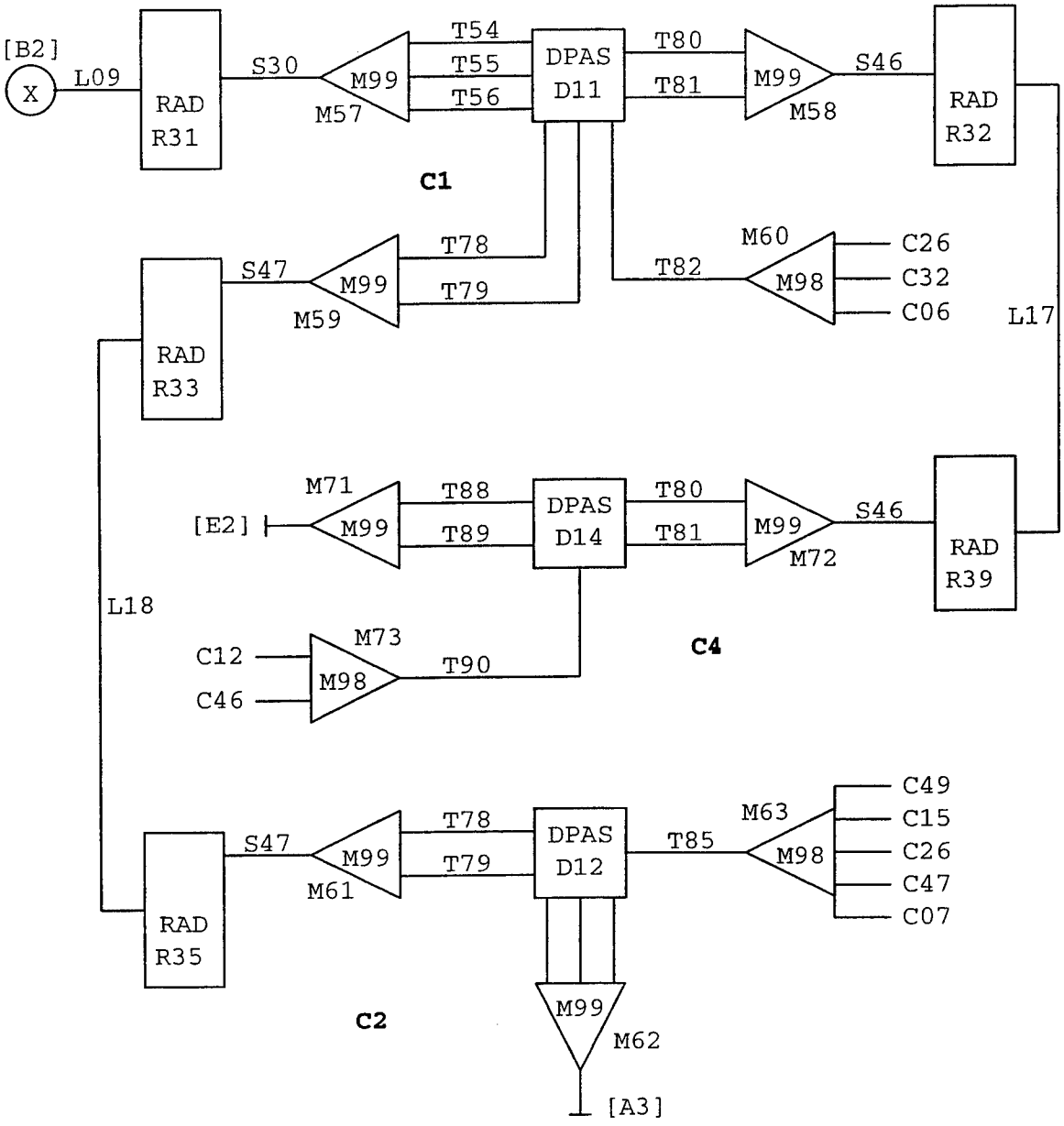


Figure 3.23: Example Communication Network (Part 2).

DPAS D02 Trunk Channel Interconnections				
Trunk	Channel	Trunk	Channel	Circuit
T14	1	T13	1	C32
T14	2	T12	1	C10
T14	3	T12	2	C03

Table 3.8: Circuit Switching Map for DPAS D02

DPAS D08 Trunk Channel Interconnections				
Trunk	Channel	Trunk	Channel	Circuit
T44	1	T57	1	C03
T45	1	T54	3	C32
T45	2	T54	4	C07
T45	3	T54	5	C12
T45	4	T59	3	C13
T45	5	T57	3	C25
T58	1	T54	2	C15
T58	2	T56	1	C47
T59	1	T57	2	C37
T59	2	T54	1	C06

Table 3.9: Circuit Switching Map for DPAS D08

DPAS D11 Trunk Channel Interconnections				
Trunk	Channel	Trunk	Channel	Circuit
T54	1	T82	3	C06
T54	2	T78	1	C15
T54	3	T82	2	C32
T54	4	T78	2	C07
T54	5	T80	1	C12
T56	1	T78	3	C47
T79	2	T80	2	C49
T82	1	T79	1	C26

Table 3.10: Circuit Switching Map for DPAS D08

DPAS D05 Trunk Channel Interconnections				
Trunk	Channel	Trunk	Channel	Circuit
T12	1	T39	3	C10
T12	2	T44	1	C03
T13	1	T45	1	C32
T39	1	T45	2	C07
T39	2	T45	3	C12
T39	4	T45	4	C13
T39	5	T45	5	C25

Table 3.11: Circuit Switching Map for DPAS D05

DPAS D09 Trunk Channel Interconnections				
Trunk	Channel	Trunk	Channel	Circuit
T57	1	T64	1	C03
T57	2	T64	2	C37
T57	3	T64	3	C25
T58	1	T64	5	C15
T58	2	T63	2	C47
T63	1	T64	4	C48

Table 3.12: Circuit Switching Map for DPAS D09

DPAS D12 Trunk Channel Interconnections				
Trunk	Channel	Trunk	Channel	Circuit
T78	1	T85	2	C15
T78	2	T85	5	C07
T78	3	T85	4	C47
T79	1	T85	3	C26
T79	2	T85	1	C49

Table 3.13: Circuit Switching Map for DPAS D12

DPAS D14 Trunk Channel Interconnections				
Trunk	Channel	Trunk	Channel	Circuit
T80	1	T90	1	C12
T80	2	T89	2	C49
T90	2	T89	1	C46

Table 3.14: Circuit Switching Map for DPAS D14

Circuit	Trunk Channel Path
C03	⟨[T14:3] [T12:2] [T44:1] [T57:1] [T64:1]⟩
C06	⟨[T59:2] [T54:1] [T82:3]⟩
C07	⟨[T39:1] [T45:2] [T54:4] [T78:2] [T85:5]⟩
C10	⟨[T14:2] [T12:1] [T39:3]⟩
C12	⟨[T39:2] [T45:3] [T54:5] [T80:1] [T90:1]⟩
C13	⟨[T39:4] [T45:4] [T59:3]⟩
C15	⟨[T64:5] [T58:1] [T54:2] [T78:1] [T85:2]⟩
C25	⟨[T39:5] [T45:5] [T57:3] [T64:3]⟩
C26	⟨[T82:1] [T79:1] [T85:3]⟩
C32	⟨[T14:1] [T13:1] [T45:1] [T54:3] [T82:2]⟩
C37	⟨[T59:1] [T57:2] [T64:2]⟩
C46	⟨[T90:2] [T89:1] ···⟩
C47	⟨[T85:4] [T78:3] [T56:1] [T58:2] [T63:2] ···⟩
C48	⟨[T64:4] [T63:1] ···⟩
C49	⟨[T85:1] [T79:2] [T80:2] [T89:2] ···⟩

Table 3.15: Trunk Channel Paths of Circuits.

Scenario: T45 input data blocked by input port of M31			
Subregion	Site	Equipment	Generated Alarm
B	B2	M42	OUTPUT-PORT-DATA-LOSS
B	B2	M45	lST-LEVEL-MUX-FAILURE
B	B3	M48	lST-LEVEL-MUX-FAILURE
C	C1	M60	lST-LEVEL-MUX-FAILURE
C	C4	M73	lST-LEVEL-MUX-FAILURE
C	C2	M63	lST-LEVEL-MUX-FAILURE

Table 3.16: Alarms Generated in Event Scenario.

The failure of this port module manifests itself as a disruption in the flow of data from trunk T45 to the multiplexer module of M31. In other words, the data from trunk T45 has been effectively blocked from being multiplexed with the input data of the remaining port modules, and is therefore not being transmitted on trunk S15. This particular internal fault of a port module occurs at a point beyond what is monitored by a MUX-99's Peripheral module. Consequently, the port module fault goes undetected and generates no TRAMCON alarms for M31. From the perspective of M31's distant-end device, MUX-99 M42, the channel in each frame of trunk S15 which transmits the input of trunk T45 contains no data whatsoever. As a result, no data from trunk T45 input to the faulty port module of M31 is output from the corresponding port module of M42. This condition is detected by the Peripheral module of M42 which then generates an OUTPUT-PORT-DATA-LOSS alarm for that particular MUX-99. It would be worthwhile to note here that an alarm of this type is ambiguous in that there is no indication as to which of the eight possible port modules in M42 are experiencing this problem.

The number of circuits affected by this loss of data from trunk T45, can be found by considering the entries in the circuit switching map of DPAS D08 of Table 3.9. A total of five circuits from trunk T45 are switched by DPAS D08 with each one terminating downstream from M42 on a different MUX-98. Since for each of these circuits there is no data, their corresponding MUX-98 BITE modules will detect that no data is present on the output side of the corresponding port module. Since the MUX-98's comprise level three of the TRAMCON hierarchy, and hence are less important, they are capable of generating only one alarm for all the fault conditions that may be detected. Then MUX-98's M45, M48, M60, M63, and M73 are all generating a 1ST-LEVEL-MUX-FAILURE alarm due to the loss of output data from at least one of the 24 possible port modules. This event scenario is summarized in Table 3.16.

The problem chosen for this example is of particular interest since the multiplexer creating the *real* network fault does not detect the condition, and there are no alarms generated within the subregion containing the multiplexer. Rather, *sympathetic* alarms are generated in two other network subregions resulting from the propagation of the *real* fault. The subsequent problem solving process will be discussed from the viewpoints of three technical controllers, positioned at the SRCF of each subregion A, B, and C which lie within the scope of the problem. From the perspective of the technical controller located in subregion A, there is no indication that a problem exists and therefore does nothing out of the ordinary for the time being.

The tech controller for subregion C is confronted with three general 1ST-LEVEL-MUX-FAILURE alarms and has no indication, within the extent of its subregion, as to what may have caused these alarms. Since a 1ST-LEVEL-MUX-FAILURE alarm does not relay any useful information, other than a particular MUX-98 is experiencing some problem, it is entirely possible the alarms are unrelated, however. The alarm data provided to the tech controllers contains information about the time at which each alarm was generated

and may reveal some probable relationships among the various alarms. In the case of this event scenario, all the alarms (in particular those of subregion C) were generated at the same time. When the tech controller makes this observation, it suggests to him the strong possibility that these alarms are all the result of a single network fault. This is but one piece of evidence which the tech controller may apply to the problem in an attempt to eliminate, or lend support to, a possible cause of the alarms in its subregion. A more general strategy employed by the tech controller is to trace the paths of each circuit which terminates on at least one of the MUX-98's generating an alarm. With this information the tech controller is able identify what these alarms have in common and formulate a set of possible network faults which have the potential for causing this particular pattern of local alarms.

Considering the network architecture of the affected sites in subregion C as shown in Figure 3.23, it can be seen that the tech controller must trace a total of nine circuits. Since the SRCF of any subregion does not have trunk connection knowledge for sites other than its own, the tech controller can only trace each circuit within the extent of its subregion. Using the circuit path information contained in Table 3.15 and observing the trunk connection boundaries imposed by the limits of subregion C, the tech controller would generate the following traces of circuit path segments:

C06 ⟨[T82:3] [T54:1] ···⟩
C07 ⟨[T85:5] [T78:2] [T54:4] ···⟩
C12 ⟨[T90:1] [T80:1] [T54:5] ···⟩
C15 ⟨[T85:2] [T78:1] [T54:2] ···⟩
C26 ⟨[T82:1] [T79:1] [T85:3]⟩
C32 ⟨[T82:2] [T54:3] ···⟩
C46 ⟨[T90:2] [T89:1] ···⟩
C47 ⟨[T85:4] [T78:3] [T56:1] ···⟩
C49 ⟨[T85:1] [T79:2] [T80:2] [T89:2] ···⟩

The tech controller is now in a position to analyze the circuit path information looking for anything which the three 1ST-LEVEL-MUX-FAILURE alarms may have in common. Perhaps one of the most noticeable features of these circuit paths, is that five of the nine circuits all ride trunk T54. In addition, for each of the three MUX-98's generating an alarm, there is at least one of these five circuits that terminates on it. Thus, the failure of trunk T54 has the potential to produce the alarms being generated in this subregion, but there are no alarms on MUX-99 M57 to support such a hypothesis. It is still possible that a suitable combination of the five circuits riding trunk T54 are all affected by a single network fault located somewhere beyond the extent of this subregion. This fact is communicated to the tech controller of subregion B in hopes that it will add support for a more informed hypothesis elsewhere in the network.

In order to automate the above process so that it can be performed by a group of distributed expert agents cooperating to solve the observed problem, this expert knowledge

must be encoded in the production rule language of TESS. While this process is somewhat routine, it requires a very large investment of manpower to do completely. We determined that such a complete encoding of real world knowledge was beyond the scope of this effort. We have demonstated (for a much simpler domain) the principles used by TESS and have shown a simple example of knowledge encoding to solve a specific problem.

Chapter 4

Distributed Reasoning

4.1 Introduction

The goal of our research is to investigate problem solving behavior that is exhibited by distributed problem solving systems. When knowledge and problem solving capacity are distributed among a group of agents, coordination of the problem solving effort is critical because the ability of the system of agents to effectively reach a solution is affected by the coordination strategy that is employed. System behavior is also affected by a number of other factors, such as the specific distribution of knowledge and the ability of the agents to share knowledge among themselves.

Most distributed problem solving systems [48] [22] [6] have a structure that incorporates a high degree of domain specific knowledge in control of the problem solving process. This makes it difficult to distinguish domain dependent aspects of problem solving behavior from those that are attributable to such factors as knowledge distribution and coordination strategy. For this reason, we have chosen to investigate problem solving in the context of a formalized domain: theorem proving. When the features of a "real world" domain are removed, it is easier to focus on behaviors that are common to many distributed problem solving environments.

In this chapter we will explore how how the relative strength of an inference rule affects distributed automated reasoning. In DARES [55] [11] [56] [61] significant analysis was done on a distributed automated reasoning system that used binary resolution as its inference rule. In DARES it was shown that a distributed reasoning system was possible in which the reasoning mechanism employed no domain specific mechanisms. DARES also showed that in using binary-resolution as an inference rule, the distribution of knowledge had little effect on the performance of the system.

One criticism of the work in DARES was that the binary resolution is a relatively weak inference rule which is inefficient and costly. We have decided to continue the work begun

in DARES by exploring how a stronger inference rule would effect the performance of the reasoning architecture proposed in DARES.

In order to accomplish this we have developed a distributed automated reasoning system, SHYRLI. When we began this research and the development of SHYRLI we identified six tasks that a set of distributed agents must be able to accomplish to reason cooperatively:

1. An agent must have a representation for expressing its knowledge.
2. An agent must be able to reason on its own local data expressed in this representation.
3. An agent must be able to judge whether its reasoning is resulting in progress towards a goal.
4. An agent must be able to formulate requests for aid in obtaining knowledge that is likely to be useful in moving its line of reasoning towards a goal.
5. An agent must be able to formulate appropriate responses to requests for aid transmitted by other agents.
6. An agent must have a coordination strategy for controlling the process of interacting with other agents in the system. Specifically, controlling when an agent decides to interact with other agents and with whom an agent interacts is essential.

To satisfy the first point our system uses the first order predicate calculus for knowledge reprentation. Since we sought to compare results with DARES, using the same knowledge reprentation as DARES made sense.

The predicate calculus satisfies one of our requirements: it is domain independent. The predicate calculus is a representation language that can be used to describe many different domains [30]. By formulating a reasoning problem about a specific domain in the predicate calculus, our system may be used to reason about that domain. As our system embodies no domain specific knowledge in how it performs its reasoning, it is completely domain independent.

To satisfy point two our system makes use of hyper-resolution, an inference rule for the predicate calculus. Although there are a number of stronger inference rules than binary resolution we settled on hyper-resolution, as it is tied to binary resolution in many ways and behaves in a similar manner. Hyper-resolution outperforms binary resolution by substantially pruning its search space. The use of hyper-resolution allowed us to build a powerful reasoning system that was comparable to DARES in its architecture, but not in its behavior.

To satisfy point three, we modified the mechanism that DARES used to judge whether outside assistance was necessary to better fit our new inference rule.

To satisfy points four and five we took advantage of the way in which hyper-resolution behaves to construct mechanisms for formulating requests for assistance and replies to requests for assistance. These mechanisms allow the agents to ask for knowledge they can use to continue reasoning, while recognizing knowledge that does not need to be communicated.

To satisfy point six, we use a broadcast mechanism. An agent broadcasts its request to all other agents when it is in need of assistance. If an agent has knowledge that fits another agent's request it responds immediately. Otherwise, it places the request to the side until it generates knowledge that fits the request. Once knowledge is generated that fits the request a reply is sent.

The development of SHYRLI has given us a mechanism that allows us to recognize what knowledge must be shared and what can be kept private in order for a group of agents to solve a problem. By not communicating irrelevant knowledge we reduce communication bandwidth and processing time. By incorporating a stronger inference rule into the architecture proposed in DARES we have increased the performance of the architecture. We have also seen that our system may be made to perform problem solving based on a functional distribution of task knowledge, and that our communication mechanisms are able to coordinate the different functional roles of a set of agents to cooperatively solve problems.

We have found that the change in inference rule has created a significant behavior difference between DARES and SHYRLI. DARES agents are essentially performing a breadth first search over a rather large search space. SHYRLI agents, although still using an inference rule that executes a breadth first search, search a significantly pruned space.

4.1.1 Related Work

There are two types of work that relate to this research. The first is work in the realm of automated reasoning using the first order predicate calculus. The second is work in the realm of distributed automated reasoning.

Much has been done in the area of automated reasoning. The key seminal work was the discovery of resolution [71]. This discovery was followed by a great deal of activity in which many different inference rules were developed. These included binary resolution [71], hyper-resolution [70], unit resolution, semantic resolution, paramodulation [69], and hyper-paramodulation. Binary resolution is a simplified form of resolution. Hyper-resolution, semantic resolution and unit resolution are attempts to increase the efficiency of resolution. Paramodulation is an inference rule that is more efficient in reasoning about equality. These inference rules were developed to operate on knowledge represented in the predicate calculus and are usually used to operate in a refutation based proving style.

A problem with each of these mechanisms is that they essentially perform their reasoning in a breadth first fashion. Their search spaces grow in an exponential fashion. As

they are given larger and larger problems, the time they take to solve the problem increase exponentially. Mechanisms have been formulated that try to control this explosive behavior by recognizing irrelevant results or redundant information and paths in the search space. These mechanisms include set of support [89], demodulation [90], subsumption [71], tautology reduction [71], and pure literal deletion.

An alternative to resolution-based inference rules has been developed by R. Boyer and J. Moore [4]. Their reasoning system does not use the typical predicate calculus representation, but instead uses a lambda calculus/LISP like representation. Their theorem prover can also find other types of proofs besides refutation based proofs, most importantly induction based proofs. Resolution based theorem provers cannot perform mathematical induction, which is necessary for some applications such as proving computer program correctness. Although the main thrust of Boyer and Moore's research was in proving program correctness, their knowledge representation and inferencing method are domain independent.

In [10] M. Cline presents a fast parallel algorithm for unification. Unification is a necessary mechanism for proving theorems in the predicate calculus. This algorithm performed unification over a set of processors. Although this algorithm is distributed, its intent was not distributed automated reasoning, but rather the improvement or speedup of reasoning using a single or centralized knowledge base. The purpose of SHYRLI is to explore how a distributed set of agents can perform distributed reasoning given a set of knowledge that is already distributed.

Distributed automated reasoning is an attempt to extend automated reasoning to a group or groups of reasoning agents. The key ingredient in distributed automated reasoning is the interaction of the agents. Given a set of agents, how does that set of agents coordinate its activity in order to arrive at common goals.

In [29] Les Gasser and Michael Huhns discuss current themes in Distributed Artificial Intelligence. Themes they describe that have relevance to the subject of distributed automated reasoning include the development of new problem solving architectures and the social behavior of a large collection of agents. An issue in which they place importance is what they term Deep Theories of Coordination. They mention that no broadly used definitions for coordination, cooperation, or interaction exist in the Distributed Artificial Intelligence Community. In the following paragraphs we will describe some of the current research in distributed automated reasoning that is trying to address this issue.

Different characteristics of distributed problem solving were discussed in work by Cammarata, McArthur and Steeb [6]. Agents in the distributed system could be characterized by their skills, their knowledge of their environment, their resources, and their appropriateness for different tasks. Strategies for cooperation were also characterized into two groups: organizational policies and information distribution policies. Organization policies dictate how a larger task should be decomposed into smaller tasks while information distribution policies dictate how cooperating agents communicate.

The authors describe four different organizational policies and provide experimental results for three of them. These three policies all involve selecting an agent to be the problem solver and letting that agent solve the problem after transmitting to the selected agent the necessary information. The interaction between the agents is only used to select which agent could best solve the problem at hand. There is no cooperative problem solving as such, only a group decision making process that selects which agent will be the problem solver. The authors provided no experimental results for the fourth policy, which involved cooperative problem solving. This differs from SHYRLI in that all SHYRLI agents can potentially participate in reasoning about the problem at hand. The authors applied their techniques to only one domain, air-traffic control, and do not discuss how it can be extended to other domains.

E. Durfee and V. Lesser proposed mechanisms that allow a group of agents to coordinate their activity through the use of partial global plans [22]. A partial global plan is a representation of how several nodes are working towards a larger goal. These mechanisms were presented in the domain specific environment of a distributed vehicle monitoring testbed. In this context a partial global plan is a skeleton of vehicle tracks through the testbed. The partial global plans were passed among agents as a way of communicating the agents' view of the world and partial solutions to the problem. The agents coordinated by planning how to fill gaps in the tracks and how to resolve discrepancies. Although the authors state that they believe their mechanisms could be extended to other domains, they do not provide an example of how this can be done. This differs from SHYRLI in the respect that SHYRLI agents do not have any view of what other agents are doing, SHYRLI agents only communicate the type of information they can potentially use, not a plan of how they are solving the problem.

In [50], C. Lewis and K. Sycara present a mechanism that allows a set of heterogeneous (i.e. agents with different specialties) to develop a shared model through which they can develop a shared perception of the problem at hand. The authors state that as a prerequisite to coordinating shared actions, agents must first coordinate to form a shared mental model of their knowledge. By constructing a shared model, or language, different specialists can then better reason about what interaction and coordination is necessary to solve a problem. This shared model is a high level structure that can be independent from the agents actual reasoning process.

Unlike both of these latter approaches, SHYRLI coordinates on a low level. A SHYRLI agent achieves its coordination by examining the syntactic structure of the predicate calculus clauses in its database.

4.1.2 Results Summary

As a tool, DARES was limited in its practicality. Its inference rule was so weak that in order to solve even small problems extensive resources were necessary. SHYRLI extends

the practicality of the architecture considerably. SHYRLI's inference rule is sufficiently powerful to solve interesting problems, rather than small toy proofs.

SHYRLI provides a baseline case of a distributed reasoning system that incorporates no domain knowledge. SHYRLI provides a testbed that can be used to test heuristics that use domain specific information to help select what to communicate and which lines of reasoning to follow. The baseline case could be used as a point of comparison between different strategies for managing coordination and interaction in order to increase system performance.

By examining the syntactic structure of its knowledge SHYRLI agents can decide if that knowledge will have no bearing on other agents progress towards a goal. This means that agents can reliably decide what knowledge can be kept private without adversely affecting the progress of the system towards a goal.

SHYRLI's inference rule realizes its performance increase by substantially pruning the search space. We have found that SHYRLI attempts to find a proof by performing a distributed search over a substantially pruned search tree. SHYRLI, unlike DARES, does not recognize a tremendous benefit by distributing the problem. A large part of DARES performance increase came from the partitions of knowledge that were created by the distribution. The distribution effectively pruned the search space. By incorporating communications heuristics that tried to maintain a beneficial distribution of knowledge the system was able to perform reasoning on the order of magnitude faster than the single agent case. SHYRLI's inference rule prunes the search space better than DARES artificial pruning. The pruning in SHYRLI is built into the control structure of the inference rule, not the distribution of knowledge in the system.

We have performed a set of experiments using SHYRLI that have shown us that different distributions of the same knowledge over a set of agents can have a significant effect on performance. We have been able to characterize these distributions along two dimensions: the amount of parallelism and the coordination necessary to solve the problem.

We have developed a predicate calculus formalism that allows a complex problem to be divided into smaller parts and the different parts distributed over a set of agents. We illustrate the application of this formalism to a simple model a digital circuit simulator. In the problem one agent knows that a device exists that performs the function of a half-adder, but does not know how the half-adder works. The device asks other agents if they understand half-adders. When an agent is found that understands half-adders, the task of simulating the half-adder is given to that agent.

The following sections will describe SHYRLI. We first give an introduction to the predicate calculus. This will provide a background necessary for understanding the rest of this chapter. The following section describes SHYRLI. We then describe the experimentation that has been done using SHYRLI and point out certain behaviors found in the SHYRLI reasoning process. Finally we summarize the significant results of the work and discuss

future directions of research.

4.2 Knowledge Representation and Inference Rules

We are exploring domain independent issues in distributed automated reasoning. We have chosen to investigate problem solving in the context of the formalized domain theorem proving. There has been a great deal of work in the area of automated theorem proving. Much of this work uses the predicate calculus as its representation language [30]. Many mechanisms for extracting information about and reasoning within a model expressed in the predicate calculus have been devised. Among them are binary resolution [71], hyper-resolution [70], unit resolution, and paramodulation [69].

This section explains the fundamentals of the basic concepts of the predicate calculus and resolution based theorem proving. We begin by defining the syntax of the predicate calculus along with an example explaining how to interpret the syntax with a semantic meaning. We then describe two inference rules for the predicate calculus, resolution and hyper-resolution. We describe ways of controlling the reasoning processes that use resolution as an inference rule. Finally a complete example of a hyper-resolution proof is given.

4.2.1 Predicate Calculus

The predicate calculus has been selected as our form of knowledge representation. In this section we will first explain the propositional calculus, then extend it to the predicate calculus.

4.2.1.1 Propositional Calculus

The predicate calculus is based in the propositional calculus. The propositional calculus concerns declarative sentences or propositions. A proposition is a declarative sentence that is either true (**T**) or false (**F**).

This simplest proposition is an atomic symbol. For example, the following atoms are propositions:

M: The power is on.

N: The switch is in the on position.

P: The computer is plugged in.

More complex propositions may be built by combining atoms with logical connectives. The five logical connectives used in the propositional logic are not ($\neg$), conjunction ($\wedge$), disjunction ($\vee$), implies ($\rightarrow$), and equivalence ($\leftrightarrow$). The truth values of the logical connectives

are described in the following table.

M	N	$\neg$M	M $\wedge$ N	M $\vee$ N	M $\rightarrow$ N	M $\leftrightarrow$ N
T	T	F	T	T	T	T
T	F	F	F	T	F	F
F	T	T	F	T	T	F
F	F	T	F	F	T	T

Figure 4.1: Truth Values of Logical Connectives.

Complex propositions are built recursively using the rules for well-formed formulas [8]:

1. An atom is a well-formed formula.
2. If **M** and **N** are well-formed formulas then ($\neg$ **M**), (**M** $\wedge$ **N**), (**M** $\vee$ **N**), (**M** $\rightarrow$ **N**), and (**M** $\leftrightarrow$ **N**) are well-formed formulas.
3. All well-formed formulas are generated by applying the above rules.

The following precedence is assigned to the logical connectives in decreasing order so that parenthesis may be dropped for ease of reading:

$\neg$, $\wedge$, $\vee$, $\rightarrow$, $\leftrightarrow$

4.2.1.2 First-Order Predicate Calculus

The propositional calculus is useful for representing certain sets of knowledge, however it is limited. For example:

M: Odeon is a computer.

N: The switch for Odeon is in the on position.

P: If the switch for a computer is in the on position then the computer is running.

In the propositional calculus it is impossible to deduce from the above well-formed formulas that Odeon is running. This is becasue that it is not possible to associate Odeon in **M** with the computer in **P**. By adding variables to the propositional calculus this becomes possible. The predicate calculus uses predicates and variables to extend the propositional calculus so it can handle situations such as the one above.

A predicate in the predicate calculus is a function that maps a set of arguments to the set {**T** **F**}. An n-place predicate accepts n arguments. Each argument accepted by

a predicate is a term. A term is either a constant, a variable, or a function. An n-place function maps n elements in the domain to a member of the domain.

Terms are defined recursively as follows:

1. A variable is a term.
2. A constant is a term
3. If f is a n-place function symbol, and t_1, ..., t_n are terms, then $f(t_1, \ldots, t_n)$ is a term.
4. All terms are generated by applying the above rules.

An atom is an n-place predicate and is constructed using the following rule [8]:

> If P is an n-place predicate and t_1, ..., t_n are terms, then $P(t_1, \ldots, t_n)$ is an atom.

Two more symbols are also added to the predicate calculus. These are the universal ($\forall$) and the existential ($\exists$) quantifiers. These symbols are used to characterize variables. If a variable, x, is employed in a universal quantifier then every instance of x within the scope of the quantifier applies to every element in the domain. If a variable, y, is employed in a existential quantifier then every instance of y within the scope of the quantifier applies to one or more elements in the domain.

A variable is considered bound if it is within the scope of a quantifier employing that variable. A variable is considered free if and only if it is not bound.

We extend the notion of a well-formed formula of the propositional calculus to the predicate calculus by adding the notion of a more complex atom as above and the following rule about quantifiers:

> If $\mathbf{F}$ is a well-formed formula and x is a free variable in $\mathbf{F}$, then $(\forall x)\mathbf{F}$ and $(\exists x)\mathbf{F}$ are well-formed formulas. x is said to be employed by the quantifier. The scope of the quantifier is F.

Quantifiers have the least precedence of all symbols in the predicate calculus.

As an example, we will now represent the statements given at the beginning of this section in the predicate calculus.

> Odeon is a computer.
>
> Odeons switch is in the on position.
>
> If a computers switch is in the on position then the computer is running.

becomes:

P: Computer(Odeon).

Q: Switch(Odeon,On).

R: $(\forall x)$(Computer(x) $\wedge$ Switch(x,On)) $\rightarrow$ Running(x).

From these well-formed formulas we can conclude:

Running(Odeon).

This can be done by substituting Odeon with the variable x in **R**.

4.2.2 Interpretations

An interpretation is an assignment of constants to specific entities in the domain being described, functions are assigned mappings from elements in that domain to an element in the domain, and predicates are assigned mappings from elements in the domain to the values of true (**T**) or false (**F**).

This can be stated formally as follows [8]:

Definition: An interpretation of a well-formed formula **F** in the first-order logic consists of a non-empty domain D, and an assignment of "values" to each constant, function symbol, and predicate symbol occurring in F as follows:

To each constant, we assign an element in D.

To each n-place function symbol, we assign a mapping from D^n to D.

To each n-place predicate symbol, we assign a mapping from D^n to {**T**, **F**}.

Well-formed formulas can be evaluated to true or false under an interpretation over a domain D using the following rules:

If the truth values of **M** and **N** are known, then the truth value of ($\neg$ **M**), (**M** $\wedge$ **N**), (**M** $\vee$ **N**), (**M** $\rightarrow$ **N**), and (**M** $\leftrightarrow$ **N**) are evaluated using table 4.1.

$(\forall x)$**M** is evaluated to true (**T**) if the truth value of **M** is evaluated to true (**T**) for every x in D, otherwise, it is evaluated to false (**F**).

$(\exists x)$**M** is evaluated to true (**T**) if the truth value of **M** is evaluated to true (**T**) for at least one x in D, otherwise, it is evaluated to false (**F**).

Using the definition of an interpretation the following terms are defined [56]:

Definition A well-formed formula is satisfiable (consistent) if and only if there exists an interpretation in which the well-formed formula evaluates to true.

For example, the following well-formed formula is consistent:

$$(\forall x)(\forall y)(\mathrm{M}(x) \rightarrow \mathrm{M}(y)).$$

By observing the logical value of the connective $\rightarrow$ in Figure 4.1 and the rules for evaluating variables in the scope of a universal quantifier it can be observed that the truth value of this well-formed formula is dependent upon the interpretation. There is an interpretation that will make this well formed formula true. Therefore this statement is consistent.

Definition A well-formed formula is unsatisfiable (inconsistent) if and only if there exists no interpretation in which the well-formed formula evaluates to true.

For example, the following well-formed formula is unsatisfiable:

$$(\forall x)(\mathrm{M}(x) \wedge \neg \mathrm{M}(x)).$$

There is no x that would enable this well-formed formula to evaluate to true (**T**).

Definition A well-formed formula is valid if and only if for all possible interpretations the well-formed formula evaluates to true.

The following well-formed formula is valid:

$$(\forall x)(\mathrm{M}(x) \rightarrow (\exists y)\mathrm{M}(y)).$$

The only way for this formula to be false would be for $\mathrm{M}(x)$ to evaluate to true and $\mathrm{M}(y)$ to evaluate to false. However this cannot be for if $\mathrm{M}(x)$ evaluates to true there exists a y (namely x) that allows $\mathrm{M}(y)$ to evaluate to true. Therefore this well-formed formula is valid.

Definition A well-formed formula is invalid if and only if there exists at least one interpretation in which the well-formed formula evaluates to false.

The example given for a consistent well for formed formula is also invalid. This is due to the fact that there is an interpretation under which that well-formed formula evaluates to false.

Definition A well-formed formula $\mathbf{P}$ is said to be a logical consequence of a set of well-formed formulas $\{\mathbf{Q}_1, \mathbf{Q}_2, \ldots, \mathbf{Q}_n\}$, if for every interpretation satisfying the set of well-formed formulas $\{\mathbf{Q}_1, \mathbf{Q}_2, \ldots, \mathbf{Q}_n\}$, the well-formed formula $\mathbf{P}$ is also satisfied.

In our earlier example about the computer named Odeon there was an implied interpretation in the well-formed formulas that we wrote. We implied the interpretation that Odeon was a computer, that On meant a state that the switch was in, that the predicate switch mapped a computer and a switch state to a value of true or false if the switch was in the state given as an argument. This is not the only interpretation of these sentences. This is easier to see if we rewrite the well-formed formulas to contain no semantic information:

P(A).

Q(A,B).

$(\forall x)(\mathrm{P}(x) \wedge \mathrm{Q}(x,\mathrm{B})) \rightarrow \mathrm{R}(x)$.

With the names of the symbols changed in the above example the original interpretation of the well-formed formulas becomes obscure. Indeed we could even map another interpretation to these same formulas.

Beverage(Beer).

In(Beer,Refrigerator).

$(\forall x)(\mathrm{Beverage}(x) \wedge \mathrm{In}(x,\mathrm{Refrigerator})) \rightarrow \mathrm{Cold}(x)$.

The implied interpretation here is completely different than the one we were considering when we wrote the formulas.

4.2.3 Unification

Unification is a process for determining whether two well-formed formulas can be made identical through the use of an appropriate substitution [8]. A substitution is a set of variables and associated terms. A substitution is defined formally as follows:

Definition A substitution is a finite set of ordered pairs of the form $\{t_1/v_1, \ldots, t_n/v_n\}$, where every v_i is a variable, t_i is a term, t_i does not contain v_i, and no two elements contain the same v_i after the stroke symbol.

Now that we have defined substitutions, we define how to apply a substitution to a well-formed formula.

Definition: Let $\theta = \{t_1/x_1, \ldots, t_n/x_n\}$ be a substitution and **E** be a well-formed formula. Then $\mathbf{E}\theta$ is a well-formed formula obtained by replacing simultaneously each occurrence of the variable x_i, $1 \leq i \leq n$, in **E** by the term t_i.

As an example consider the substitution θ and the well-formed formula **R**:

$\theta = \{\text{Odeon}/x\}$

R: (Computer(x) $\wedge$ Switch(x,On)) $\rightarrow$ Running(x).

Then $\mathbf{R}\theta$ is:

$\mathbf{R}\theta$ = (Computer(Odeon) $\wedge$ Switch(Odeon,On)) $\rightarrow$ Running(Odeon).

Unification finds a unifying substitution θ for a set of well-formed formulas. When θ is applied to each member of the set of well-formed formulas every member in the set is identical. This is defined formally as follows:

Definition: A substitution θ is called a unifier for a set of well-formed formulas $\{\mathbf{E}_1, \ldots, \mathbf{E}_k\}$ if and only if $\mathbf{E}_1\theta = \cdots = \mathbf{E}_n\theta$. The set of well-formed formulas $\{\mathbf{E}_1, \ldots, \mathbf{E}_k\}$ is said to be unifiable if there is a unifier for that set.

The unifier of the following two well-formed formulas is the substitution θ given above:

P: Computer(Odeon).

R: Computer(x).

When θ is applied to either of these well-formed formulas the result will be identical with **P**.

Through the course of computing a unification it is sometimes necessary to combine two substitutions. The composition of two substitutions is defined as follows:

Definition If $\theta = \{t_1/x_1, \ldots, t_n/x_n\}$ and $\lambda = \{u_1/y_1, \ldots, u_n/y_n\}$ are two substitutions, then the composition of θ and λ is represented by $\theta\lambda$ and has the form:

$$\{t_1\lambda/x_1, \ldots, t_n\lambda/x_n, u_1/y_1, \ldots, u_n/y_n\}$$

and by deleting any element of the form:

$t_i\lambda/x_j$ where $t_i\lambda = x_j$

u_i/y_j where $y_j \in \{x_1, \ldots, x_n\}$

As an example assume we have the substitutions σ and γ where:

$$\sigma = \{\text{Odeon}/x, \text{F}(z,\text{On})/y\, u/v\}$$

$$\gamma = \{\text{Odeon}/z, v/u, \text{On}/x\}$$

Then:

$$\{\text{Odeon}\gamma/x, \text{F}(z,\text{On})\gamma/y\, u\gamma/v, \text{Odeon}/z, v/u, \text{On}/x\} =$$

$$\{\text{Odeon}/x, \text{F}(\text{Odeon},\text{On})/y\, v/v, \text{Odeon}/z, v/u, \text{On}/x\}$$

Two of the members of this set have to be removed because they meet the requirements for deletion in the definition of substitution composition. Thus the composition of σ and γ is:

$$\{\text{Odeon}/x, \text{F}(\text{Odeon},\text{On})/y\, \text{Odeon}/z, v/u\}$$

In many situations there is more than one θ that can unify a set of expressions. The most general unifier has the property that it is unique for a set of expressions up to variable renaming. The most general unifier is defined as follows:

Definition: A unifier σ for a set of well-formed formulas $\{\mathbf{E}_1, \ldots, \mathbf{E}_k\}$ is a most general unifier if and only if for each unifier θ for the set there is a substitution λ such that θ equals the composition of σ and λ.

4.2.4 Skolem Normal Form

In order for certain inference rules based in the predicate calculus to work properly, the predicate calculus sentences must be in a normal form. The Skolem normal form was introduced in [17]. The Skolem normal form is based on two other normal forms, conjunctive normal form and prenex normal form, along with the concept of Skolem functions.

A well-formed formula is said to be quantifier free if it contains no instances of the universal ($\forall$) or the existential ($\exists$) quantifiers. A well-formed formula is said to be in prenex normal form if it is of the form:

$$(Q_1 x_1) \ldots (Q_n x_n)(F)$$

where $n \geq 1$, Q_i is either the universal ($\forall$) or the existential ($\exists$) quantifier, and $\mathbf{F}$ is quantifier free.

We begin our definition of *conjunctive normal form* by defining literals and clauses.

Definition: A literal is an atom or a negated atom.

Definition: A clause is a disjunction of literals. The well-formed formulas $L_1 \vee L_2 \vee \cdots \vee L_n$ where $L_1, L_2, \ldots, L_n$ are literals is a clause.

A clause can be thought of as a set of literals with the disjunction implied.

Conjunctive normal form can now be defined in terms of clauses:

Definition: *Conjunctive normal form* is a conjunction of clauses.

Replacing existential quantifiers with functions may be done without affecting consistency. This process is called skolemization. Skolemization can best be presented with an example. Assume we have the well-formed formula:

$$(\exists v)(\forall w)(\exists x)(\forall y)(\exists z)P(v, w, x, y, z)$$

The skolemization of this well-formed formula is as follows:

$$(\forall w)(\forall y)P(f_1, w, f_2(w), y, f_3(w, y))$$

The first existential quantifier in the first expression states that there exists one or more objects in the domain that makes the five-place predicate P true. We simply instantiate a Skolem constant f_1 that corresponds to one of those objects. The next existential quantified variable, x, is preceded by the universally quantified variable w. Depending on the predicate P the value of x may depend on the value of w. Thus we provide the one-place function f_2 to provide this mapping. In the same manner we provide the two-place function f_3 for the existentially quantified variable z because z may depend on the two universally quantified variables w and y.

A well-formed formula is said to be in Skolem standard form if the expression is of the form:

$$(Q_1 x_1) \ldots (Q_n x_n)(F)$$

where every Q_i is the universal quantifier and **F** is quantifier free and in conjunctive normal form.

To convert a well-formed formula into Skolem normal form the following steps are followed:

1. Convert the well-formed formula into prenex normal form.
2. Convert the quantifier free portion of the prenex normal form into conjunctive normal form.
3. Perform skolemization to eliminate existential quantifiers.

4.2.5 Resolution

Resolution [71] is a procedure for performing proof by refutation. Resolution works with a set of well-formed formulas that are in Skolem normal form. Every variable in Skolem normal form is universally quantified, therefore the quantifiers may be assumed. The set of knowledge that resolution works with is a conjunction of clauses. As one further step the variables in each clause are renamed so that they are unique.

Consider a well-formed formula **P**. We would like to show that **P** is valid. Under the definition of a valid well-formed formula, for **P** to be true it must evaluate to true (**T**) under all interpretations. To prove that **P** is valid we need only evaluate **P** under all interpretations. If **P** always evaluates to true **T** we know that **P** is valid. As the number of symbols in a well-formed formula grows, however, the number of interpretations increases exponentially.

If **P** is valid then $\neg(\mathbf{P})$ evaluates to false under all interpretations. $\neg(\mathbf{P})$ is said to be unsatisfiable. If we can show that $\neg(\mathbf{P})$ is unsatisfiable then we can assume **P** is valid. This is the goal of resolution, and is known as a refutation proof. Remember that our knowledge is in the form of a conjunction of disjunctions, where every disjunction is called a clause. If we can find two complementary clauses $\neg Q$ and Q then the set of clauses must be false under all interpretations. Under any arbitrary interpretation one of the two clauses will always be false, and under the truth value of a conjunction the entire set will be false. Resolution attempts to find two such clauses, and if it cannot, it attempts to deduce clauses of this type from the set of clauses.

To get an intuitive idea of how resolution works consider the following example. Assume the following clauses are known to be true under any interpretation:

P: $(A \vee C)$

Q: $(B \vee \neg C)$

We can then deduce that the following clause is true:

R: $(A \vee B)$

We can make this deduction because of the complementary literal C in **P** and **Q**. We know that C is either true or false. If C is true then B must be true. If C is false then A is true. So either A is true or B is true, thus clause **R**.

This is formalized in the following definition:

Definition: Let $\mathbf{C}_1$ and $\mathbf{C}_2$ be two clauses with no variables in common. Let L_1 and L_2 be two literals in $\mathbf{C}_1$ and $\mathbf{C}_2$ respectively. If L_1 and $\neg L_2$ have a most general unifier σ, then the clause:

$$(\mathbf{C}_1\sigma - L_1\sigma) \cup (\mathbf{C}_2\sigma - L_2\sigma)$$

is called a binary resolvent of $\mathbf{C}_1$ and $\mathbf{C}_2$.

Let **P** and **Q** be clauses such that:

P: A(x,y) $\vee$ C(x).

Q: B(z) $\vee$ $\neg$C(A).

Remember that clauses may be interpreted as sets of literals, thus:

P: {A(x,y), C(x)}.

Q: {B(z), $\neg$C(D)}.

Clauses **P** and **Q** contain the complementary literal C. To unify C(x) and C(D) we compute the substitution σ:

σ: {D/x}.

Using C as the complementary literal a resolvent of **P** and **Q** is:

$$(\{\text{A}(x,y),\ \text{C}(x)\}\sigma - \{\text{C}(x)\}\sigma) \cup (\{\text{B}(z),\ \neg\text{C(D)}\}\sigma - \{\neg\text{C(D)}\}\sigma)$$

Which simplifies to:

{A(D,y), B(z)}

The goal of resolution, when applied to refutation proofs, is to deduce a contradiction ($\Box$). A contradiction is known as an empty clause and occurs when two complementary clauses $\neg Q$ and Q are resolved.

Some proof procedures will not necessarily find a proof for a certain set of well-formed formulas even if one exists. If a proof procedure is always guaranteed to find a proof if one exists it is said to be complete. Completeness can be defined as follows:

Definition: An inference rule is complete if and only if given a set of unsatisfiable clauses S, the inference rule can deduce that the set S is unsatisfiable.

There are certain cases that if a contradiction exists, it is not necessarily the case that binary resolution will find the contradiction. For example, binary resolution will never infer a contradiction from the following clauses:

P: Q(x) $\vee$ Q(y).

Q: $\neg Q(v) \vee \neg Q(w)$.

By itself, binary resolution is not complete. When combined with factoring however, it is complete. Factoring is defined as follows:

Definition: If two or more literals of a clause **C** have a most general unifier σ, then $\mathbf{C}\sigma$ is called a factor of **C**.

The clauses **P** and **Q** above can be factored into the two clauses **P'** and **Q'**:

P': $Q(x)$.

Q': $\neg Q(v)$.

From these clauses a contradiction is easily deduced.

A resolvent of two clauses $\mathbf{C}_1$ and $\mathbf{C}_2$ is defined as:

Definition: A resolvent of two clauses $\mathbf{C}_1$ and $\mathbf{C}_2$ is a binary resolvent of $\mathbf{C}_1'$ and $\mathbf{C}_2'$ where $\mathbf{C}_1'$ is $\mathbf{C}_1$ or a factor of $\mathbf{C}_1$ and $\mathbf{C}_2'$ is $\mathbf{C}_2$ or a factor of $\mathbf{C}_2$.

Using the above rule to generate resovents, resolution is complete [8].

One problem with resolution-based theorem provers is that they are not guaranteed to halt. Let S be a valid set of clauses. Assume that the number of resolvents that may be generated from S is infinite. Any theorem prover that is applied to the set S will generate resolvents forever.

We will now present a small example of resolution. Remember the example of the computer named Odeon:

P: Computer(Odeon).

Q: Switch(Odeon,On).

R: $(\forall x)(\text{Computer}(x) \wedge \text{Switch}(x,\text{On})) \rightarrow \text{Running}(x)$.

The hypothesis we would like to prove is that Odeon is running:

H: Running(Odeon).

We negate this statement and add it to our list of statements

The next step is to convert these predicate calculus statements into Skolem normal form:

C1: Computer(Odeon).

C2: Switch(Odeon,On).

C3: $\neg$Computer(x) $\vee$ $\neg$Switch(x,On) $\vee$ Running(x).

H1: $\neg$Running(Odeon).

It is not necessarily the case that each predicate calculus statement will generate only one clause.

We can now begin performing resolution. By resolving **H1** and **C3** we get:

R1: $\neg$Computer(Odeon) $\vee$ $\neg$Switch(Odeon,On).

We can then resolve **R1** with **C1**:

R2: $\neg$Switch(Odeon,On).

Finally we can resolve **R2** with **C2**:

R3: $\square$.

This results in a contradiction indicating that our hypothesis was indeed valid.

4.2.5.1 Hyper-Resolution

Resolution based theorem provers are computationally explosive because they tend to produce a great many clauses that have no bearing on the proof or are redundant in nature. Many mechanisms for reducing the amount of redundant information have been proposed. Among them are deletions strategies which will be discussed in a later section. Another method is to try to produce fewer redundant clauses. Hyper-resolution accomplishes this [70].

Hyper-resolution is a form of semantic resolution. Semantic resolution reduces the number of possible resolutions by dividing the set of clauses into two sets and only allowing resolutions between those two sets. Semantic resolution uses an interpretation to decide which clauses are assigned to each set. Any interpretation may be used. If a clause is true (**T**) under the interpretation it is assigned to the set of *nuclei*, if it is false (**F**) it is assigned to the set of *electrons*.

The division into two sets limits the number of resolvents, because this limits the combinations of clauses that may be resolved together. Semantic resolution limits the number of resolvents further by imposing an ordering on the predicate symbols. When resolving between these two sets of clauses we can only resolve with the literal that contains the largest predicate symbol in the set of electrons under the ordering.

Semantic resolution is defined formally as follows [8]:

Let I be an interpretation. Let P be an ordering of predicate symbols. A finite set of clauses $\{\mathbf{E}_1, \ldots, \mathbf{E}_q, \mathbf{N}\}$, $q \geq 1$, is called a *semantic clash* with respect to P and I, if and only if $\mathbf{E}_1, \ldots, \mathbf{E}_q$ (called *electrons*) and $\mathbf{N}$ (called the *nucleus*) satisfy the following conditions.

1. $\mathbf{E}_1, \ldots, \mathbf{E}_q$ are false in I.
2. Let $\mathbf{R}_1 = \mathbf{N}$. For each $i = 1, \ldots, q$, there is a resolvent $\mathbf{R}_{i+1}$ of $\mathbf{R}_i$ and $\mathbf{E}_i$.
3. The literal in $\mathbf{E}_i$, which is resolved upon, contains the largest predicate symbol in $\mathbf{E}_i$
4. $\mathbf{R}_{q+1}$ is false in I.

$\mathbf{R}_{q+1}$ is called a *PI-resolvent.*

Consider an interpretation I in which every atom in the set of clauses is false under I. Under this interpretation every *PI-resolvent* must contain only positive literals. This is hyper-resolution and the *PI-resolvent* is called a hyper-resolvent. With the addition of factoring, hyper-resolution is complete [8].

As an example we will show how hyper-resolution may be used to deduce that our computer Odeon is running. Remember the Skolem normal form of our predicate calculus statements and our negated hypothesis:

C1: Computer(Odeon).

C2: Switch(Odeon,On).

C3: $\neg$Computer(x) $\vee$ $\neg$Switch(x,On) $\vee$ Running(x).

H1: $\neg$Running(Odeon).

The first step is to choose an ordering P. For simplicity we will choose alphabetical order by predicate name. The next step is to divide the clauses into the sets of electrons and nuclei.

Electrons:

C1: Computer(Odeon).

C2: Switch(Odeon,On).

Nuclei:

C3: $\neg$Computer(x) $\vee$ $\neg$Switch(x,On) $\vee$ Running(x).

H1: $\neg$Running(Odeon).

We can now begin generating hyper-resolvents. We can resolve **C1**, **C2**, and **C3** together to generate the hyper-resolvent:

HR1: Running(Odeon).

HR1 may be hyper-resolved with **H1** to generate the hyper-resolvent:

HR2: □.

We have generated a contradiction and therefore shown that our hypothesis was valid.

4.2.6 Deletion Strategies

Another method of reducing the amount of redundant information is the use of deletion strategies. A deletion strategy is used to delete clauses that are redundant from the set of clauses. Not only does this reduce the current number of clauses, but it prevents the deduction of further redundant clauses from the current redundant information. Three deletions strategies we will describe are subsumption, demodulation, and tautology reduction.

4.2.6.1 Subsumption

Subsumption was first proposed as a deletion strategy for resolution based theorem provers in the paper that introduced resolution [71]. Subsumption is used to eliminate clauses that are less general than other clauses. If there is a clause that contains general information, any clause that is a redundant specification of that information is unimportant. For example, if I know that all dogs have four legs, knowing that fido is a dog who has four legs is redundant.

The formal definition of subsumption is:

Definition: If **C** and **D** are two distinct nonempty clauses, we say that **C** subsumes **D** if there is a substitution σ such that $\mathbf{C}\sigma \subseteq \mathbf{D}$.

Subsumption algorithms are expensive and have been shown to be NP-complete [34]. It has been argued that the gains in efficiency that the reduction of the subsumed clauses provides is outweighed by the cost of the subsumption algorithm itself. Most of the successful theorem provers, however, employ subsumption [34].

4.2.6.2 Demodulation

Demodulation, in part, consists of rewriting a set of clauses into a canonical form. As an example, suppose it is known that inv(inv(x)) is equivalent to x. Given two clauses:

A1: P(inv(inv(x))).

A2: $\mathrm{P}(x)$.

we can reduce **A1** to $\mathrm{P}(x)$ and then delete it because the information is already present in our set of clauses.

Demodulation is described formally as [90]:

Definition: Let W be a finite set of the form $\{\alpha_1\beta_1, \ldots, \alpha_n\beta_n\}$ where α_i and β_i are terms. A *modulant* relative to W of a clause $\mathbf{A}$ consists of replacing a single occurrence of a term α' in $\mathbf{A}$ with the term $\beta_i\sigma$ where σ is a substitution and $\alpha_i\sigma$ is identical to α' and $\alpha_i\beta_i$ is a member of W.

Definition Let W be a finite set of the form $\{\alpha_1\beta_1, \ldots, \alpha_n\beta_n\}$ where α_i and β_i are terms. *Demodulation* consists of replacing $\mathbf{A}$ by a modulant $\mathbf{C}$ of $\mathbf{A}$ relative to W. $\mathbf{C}$ is determined by generating a sequence $\mathbf{A}_0$, ..., $\mathbf{A}_k$ such that $\mathbf{A}$=$\mathbf{A}_0$, $\mathbf{C}$=$\mathbf{A}_k$, $\mathbf{A}_{i+1}$ is a modulant of $\mathbf{A}_i$ relative to W, and $\mathbf{A}_k$ has no modulant relative to W.

4.2.6.3 Tautology Reduction

Tautology reduction deletes clauses that are true under every interpretation from the set of clauses. If a clause is true under every interpretation it is valid, and therefore will not be part of the contradiction. Removing a tautology from the database will not remove any information necessary in finding a contradiction.

For example consider the clause:

C1: $\mathrm{M}(x) \vee \neg\mathrm{M}(x)$.

Any other clause, **C2**, that resolves with **C1** will generate a resolvent that is identical to clause **C2**. This is due to the fact that in unifying the variable x and removing the resolved upon literals in **C1** and **C2**, the remaining literal in **C1** will replace the literal that was resolved upon in **C2**. Therefore no useful work will be accomplished.

As another example consider the following clause:

C3: $\neg\mathrm{M}(x) \vee \mathrm{M}(x) \vee \mathrm{P}(y)$.

This clause can be stated as: if $\mathrm{M}(x)$ is true then $\mathrm{M}(x)$ or $\mathrm{P}(y)$ is true. By resolving with this clause we will produce redundant information. If we know that $\mathrm{M}(x)$ is true, it is redundant to know that $\mathrm{M}(x)$ or $\mathrm{P}(y)$ is true. A resolvent of **C3** is the type of information that will be deleted through the use of subsumption. Therefore, deleting **C3** from our database of clauses is justified.

4.2.7 Control Strategies

The way in which an inference rule is applied can contribute enormously to the efficiency and performance of a reasoning system. Hyper-resolution can be seen as a control strategy for resolution. Hyper-resolution restricts and directs how resolution may be applied to the clauses. We will describe three additional control strategies in this section. The first, level saturation, is used by SHYRLI. The second, set of support is a simple but powerful extension of level saturation. Finally we will discuss unit preference.

4.2.7.1 Level Saturation

Level saturation is a brute force, breadth first search control strategy for resolution. Assume the set of clauses known is S. Level saturation works by generating sets S_0, S_1, ..., S_n where [8]:

$$S_0 = S,$$

$$S_n = \{\text{resolvents of } \mathbf{C}_1 \text{ and } \mathbf{C}_2 \text{ where } \mathbf{C}_1 \in (S_0 \cup \cdots \cup S_{n-1}) \text{ and } \mathbf{C}_2 \in S_{n-1}\}.$$

This process of generating new sets continues until a contradiction is generated, in which case a proof has been found, or until no new clauses can be generated, in which case the hypothesis is invalid. Each S_k is called a saturation level, in this case saturation level k. In certain cases the number of saturation levels is infinite, in which case a resolution based reasoning system might never halt.

4.2.7.2 Set of Support

When performing refutation based proofs the hypothesis usually does not stand by itself. It is often the case that we know that clauses $\mathbf{C}_1, \ldots, \mathbf{C}_n$ are consistent, and based on these clauses we are trying to conclude the validity of H. In a refutation based proof we would attempt to show that $\mathbf{C}_1, \ldots, \mathbf{C}_n, \neg H$ is unsatisfiable. Since $\mathbf{C}_1, \ldots, \mathbf{C}_n$ are consistent, any clauses deduced from these clauses should also be consistent and will not contain the contradiction. The set of support strategy attempts to increase the efficiency of the reasoning system by avoiding resolving clauses in $\mathbf{C}_1, \ldots, \mathbf{C}_n$ with each other [89].

The set of support strategy works by dividing the set of clauses into two sets. The set of support T, and the rest of the clauses not in the set of support, $(S - T)$. In most cases the clauses making up the negated hypothesis are chosen as the set T. Set of support only allows clauses to be resolved between the two sets. Any new resolvent is placed in the set T. There are many similarities between set of support and hyper-resolution. The two have been proven to be two instances of semantic resolution [74].

Set of support is easily implemented using the level saturation control strategy. By simply setting $S_0 = S - T$ and $S_1 = T$ and generating the sets $S_2 \ldots S_n$ in the same manner described for level saturation the rules for set of support will be satisfied.

4.2.7.3 Unit Preference

The unit preference strategy attempts to reduce the search space of a resolution based refutation proof by recognizing that some resolvents may be closer to a contradiction than others [88]. The resolvent of two single literal clauses is a contradiction, therefore unit preference attempts to resolve single literal clauses together first. Unit preference works by sorting the clause list of each saturation level so that single literal clauses are considered first, allowing a potential contradiction to be deduced as soon as possible. When a contradiction is found the theorem prover may stop processing.

4.2.8 An Example

Now that we have defined the predicate calculus and mechanisms for reasoning about knowledge represented in the predicate calculus we will present an example of a complete hyper-resolution proof. We hope that this will provide an example of how the concepts presented in the previous section are tied together.

We have implemented the theorem proving component of SHYRLI using hyper-resolution as an inference rule. SHYRLI 's theorem prover can be configured to use tautology deletion and subsumption. SHYRLI's theorem prover utilizes a saturation level control strategy. Its algorithm in generating resolvents, although not exactly unit preference, generates resolvents of smaller nuclei first.

The proof we will present here was generated by SHYRLI using one theorem proving agent.

The clauses used in the proof are:

A1:	$I(y) \vee \neg F(z) \vee \neg P(x) \vee \neg Q(x)$	(Axiom)
A2:	$\neg P(x) \vee \neg T(x)$	(Axiom)
A3:	$P(y) \vee \neg S(A,y)$	(Axiom)
A4:	$J(x) \vee \neg I(C)$	(Axiom)
A5:	$\neg J(A)$	(Axiom)
A6:	$\neg H(x)$	(Axiom)
A7:	$F(C)$	(Axiom)
A8:	$G(D)$	(Axiom)
A9:	$H(B) \vee R(A)$	(Axiom)
A10:	$P(A) \vee \neg G(x)$	(Axiom)
A11:	$Q(E(x)) \vee T(x) \vee \neg R(x)$	(Axiom)
A12:	$S(x,E(x)) \vee T(x) \vee \neg R(x)$	(Negated Theorem)

Our first step is to decide on an ordering of predicate symbols. To make this simple, we will choose alphabetical order by the predicate symbol. Our next step is to divide this set of clauses into the sets of nuclei and electrons. Let the interpretation I be $\{\neg F(x), \neg G(x), \neg H(x), \neg I(x), \neg J(x), \neg P(x), \neg Q(x), \neg R(x), \neg S(x, y), \neg T(x)\}$. Remember that the nuclei are clauses the are true (**T**) under I, while the electrons are false (**F**) under I. Any clause which contains a negated literal is therefore true and a nucleus, all others are electrons.

The two sets are then:

Nuclei = {**A1, A2, A3, A4, A5, A6, A10, A11, A12**}

Electrons = {**A7, A8, A9**}

The following is each saturation level and the electrons that were derived in each:

Saturation Level 1:

A13:	R(A)	Parents: **A9**, **A6**.
A14:	P(A)	Parents: **A8**, **A10**.

Saturation Level 2:

A15:	S(A,E(A)) $\vee$ T(A)	Parents: **A13**, **A12**.
A16:	Q(E(A)) $\vee$ T(A)	Parents: **A13**, **A11**.

Saturation Level 3:

A17:	P(E(A)) $\vee$ T(A)	Parents: **A15**, **A3**.

Saturation Level 4:

A18:	I(x) $\vee$ T(A)	Parents: **A16**, **A7**, **A17**, **A1**.

Saturation Level 5:

A19:	J(x) $\vee$ T(A)	Parents: **A18**, **A4**.

Saturation Level 6:

A20:	T(A)	Parents: **A19**, **A5**.

Saturation Level 7:

A21:	$\square$	Parents: **A14**, **A20**, **A2**.

Figure 4.2 shows the deduction tree for this example. Circles represent given axioms and the negated theorem, boxes represent the derived clauses. The electrons that were derived earliest are higher in the tree than electrons that were derived later. No redundant or unnecessary clauses were produced in this example so no deletion strategies were needed. This is not usually the case.

This example was originally used in the first general behavior experiment in [56]. The theorem prover used in [56] used binary resolution and set of support. The theorem prover proved the theorem in eleven saturation levels, took 713 seconds and produced 383 resolvents. Using hyper-resolution, our theorem prover proved the theorem in seven saturation levels, took eight seconds and produced nine hyper-resolvents. This is an indication of the performance benefit that can be gained through the use of hyper-resolution.

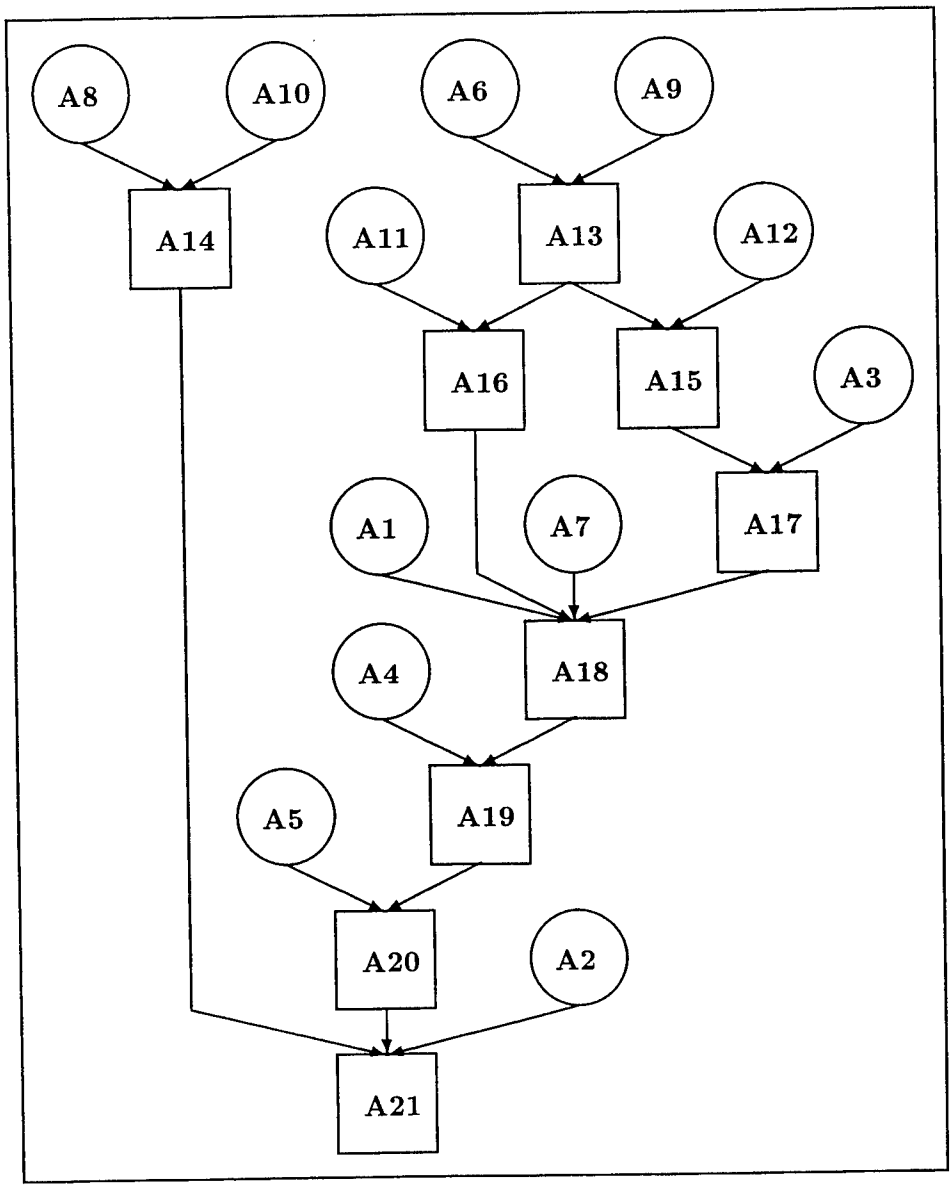


Figure 4.2: Deduction tree for the single agent example.

4.3 SHYRLI

The focus of this section is distributed hyper-resolution. We have chosen the predicate calculus and the refutation based theorem proving inference rule hyper-resolution to satisfy the first two goals outlined in the first section of this chapter. Hyper-resolution, and resolution, as originally formulated, requires that an agent have a complete world view of the problem. In distributed environments, no agent has a complete world view, as information is distributed among many agents. This can be seen in Figure 4.3. Although the entire system contains a complete world view of the domain, each agent only as a partial view. Clearly, modification of the reasoning strategy is required for distributed problem solving.

One strategy for distributed automated reasoning using the predicate calculus was implemented in DARES, the system described in [11] [55] [56] [61]. Experience with DARES demonstrated that binary resolution coupled with a set of coordination strategies and heuristics may be used to prove theorems in a system of semi-autonomous agents over which a knowledge base is distributed. One interesting aspect of DARES' behavior is that the distribution of axioms across agents seldom has strong adverse affects on DARES' performance.

Unfortunately, the inference rule used by DARES (binary resolution) is somewhat inefficient and costly. In order to better investigate problem solving behavior in distributed problem solving systems we have implemented SHYRLI. We have attempted to adapt DARES strategies to hyper-resolution. Some of the strategies carried over with little or no changes, for example the heuristic that determines whether outside assistance is necessary. Others had to be discarded and new approaches taken.

This section will describe how SHYRLI meets the six requirements for distributed reasoning. We will describe the coordination strategies used by SHYRLI and the extensions made to the heuristics used in [56] that determine whether outside assistance is necessary.

4.3.1 Distributed Hyper-Resolution

In order to satisfy the first two requirements for a distributed reasoning system we have chosen the predicate calculus as our representation language and hyper-resolution as our inference rule. We have chosen the predicate calculus because it allows a domain independent representation and we are exploring domain independent issues in distributed automated reasoning.

Before a distributed hyper-resolution theorem prover may begin reasoning the different reasoning agents must agree on an ordering P. If the agents do not agree on the ordering P they would be searching different paths throught the search space, paths which potentially would not allow an existing contradiction to be inferred. As will be seen, the P that the agents agree upon must only be a partial ordering of predicate symbols. For the time being assume that all agents share a complete ordering P of predicate symbols.

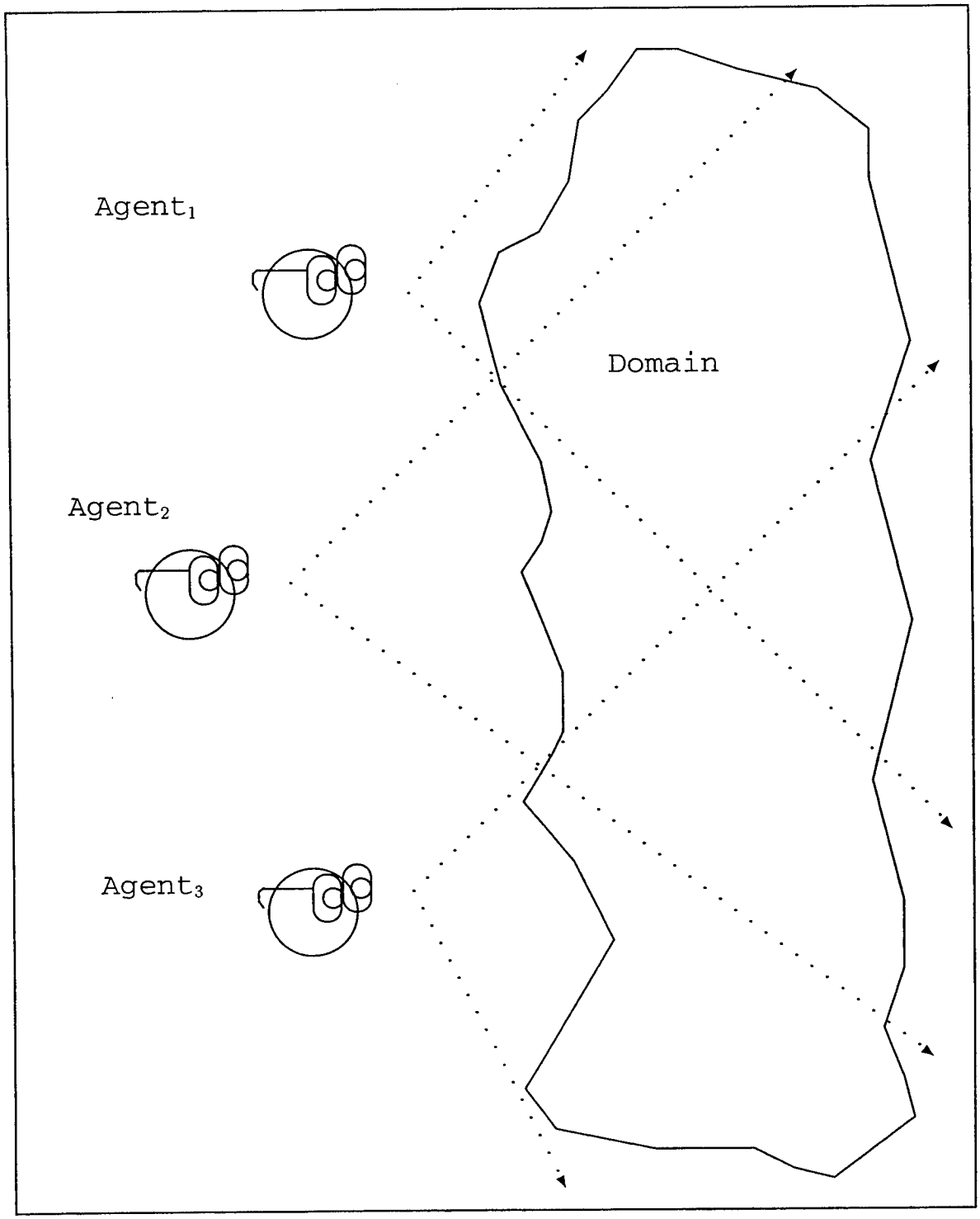


Figure 4.3: Different Agents see Different Parts of the Domain

4.3.2 SHYRLI Control Structure

SHYRLI is a system of cooperating agents. A model of the structure of SHYRLI can be seen in Figure 4.4. The SHYRLI architecture is one of several agents connected by a communication network. Agents communicate with one another via an exchange of messages over this network.

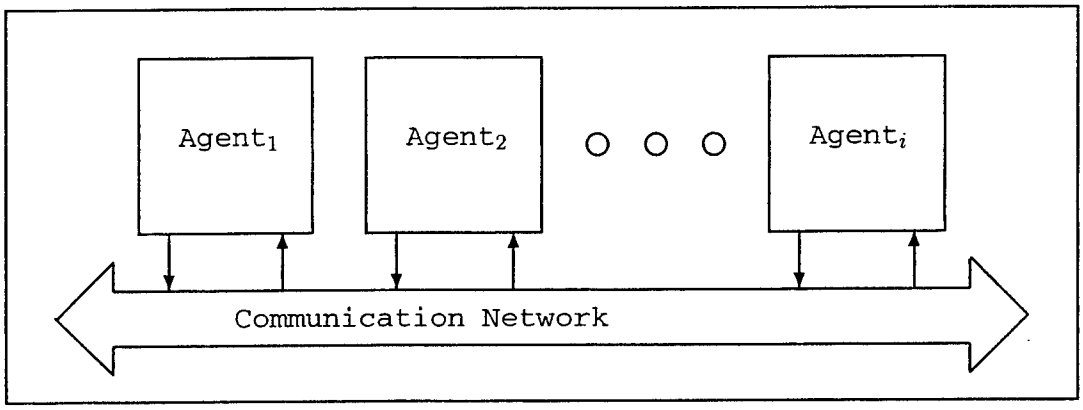


Figure 4.4: SHYRLI Agent Network

SHYRLI agents incorporate hyper-resolution in a saturation level theorem proving algorithm. Each agent is capable of performing its own problem solving using hyper-resolution as its inferencing mechanism. Each agent consists of two independent processes as show in in FIgure 4.5; the theorem prover and the secretary. The theorem prover performs the reasoning tasks necessary for individual problem solving. The theorem prover is augmented with the ability to assess local progress and formulate requests. The secretary manages communication with other agents. The secretray broadcasts the requests formulated by the theorem prover to other agents and answers requests posed by other agents.

We have formulated our agents using this two process strategy in order to avoid the deadlock condition where two theorem provers wait for responses from each other indefinitely. Let **A** and **B** be two theorem provers, each having only one thread of control. In the cycle of an inference step the theorem prover performs the following three steps:

1. Ask questions of other theorem provers.
2. Wait for responses from other theorem provers.
3. Answer questions of other theorem provers.

The theorem prover performs these steps in the order given. Imagine the scenario where theorem prover **A** and **B** simultaneously ask each other a question. They will both wait in

step 2 indefinately. The division of the theorem prover into two processes has allowed us to overcome this deadlock condition, and acheive a faster response time to questions.

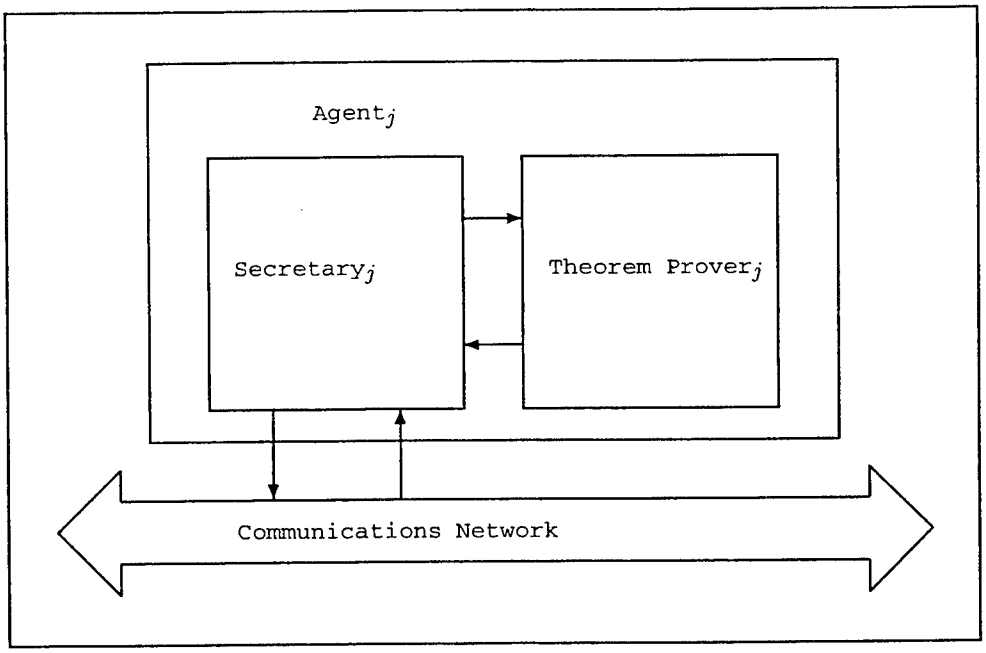


Figure 4.5: Internal View of a SHYRLI Agent

4.3.2.1 Theorem Prover Control Structure

The theorem prover process in a SHYRLI agent is responsible for local reasoning, formulation of requests, and judging whether outside assistance is necessary. The control loop for the theorem prover component is shown in figure 4.6.

The first step a theorem prover takes is to determine whether outside assistance is necessary. This is accomplished through the use of a heuristic derived from one used in DARES. If the heuristic decides that aid is needed the theorem prover formulates a request for information. The way a request is formulated is discussed in section 4.3.4.

The next step a theorem prover takes is to check whether it has received any new information from other agents. If this is the case the new knowledge is incorporated into the agents knowledge base at the current saturation level. The information is tagged with the name of the sending agent so that knowledge of the source of information can be maintained.

At this point hyper-resolution is applied and a new saturation level is generated. If a contradiction is discovered the theorem prover indicates that this is so and stops processing.

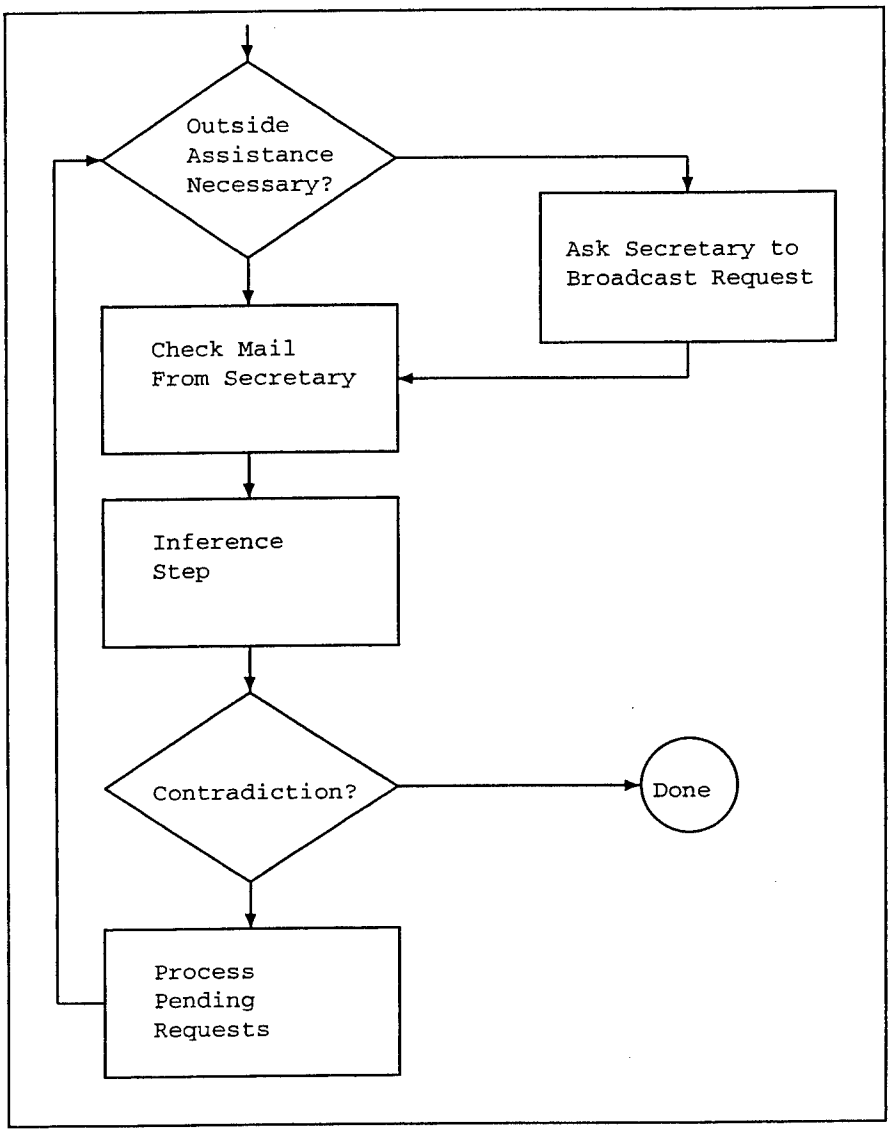


Figure 4.6: SHYRLI Theorem Prover Control Structure.

If a contradiction has not been found the agent signals the secretary that new information has been derived and the process is repeated.

4.3.2.2 Secretary Control Structure

The secretary process of a SHYRLI agent is responsible for the communication with other agents. This includes broadcasting requests to other agents in the system and answering requests posed by other agents. The control loop for the secretary process can be seen in figure 4.7.

The secretary begins by waiting for mail. This includes signals from the theorem prover that new knowledge has been derived. Once mail arrives the secretary begins processing the incoming messages.

If an incoming message is a request for aid from the associated theorem prover, the secretary broadcasts the request to the other agents it knows in the system.

If an incoming message is a request for aid from another agent, the secretary begins by constructing a reply. The way a reply is constructed is discussed in section 4.3.4. If the constructed reply is empty the secretary places the request in a buffer for future consideration, otherwise the reply is sent to the requesting agent. Any information sent to another agent is tagged with the name of the requesting agent so that any future requests from the same agent will not be answered with redundant information. If the reply is empty, but a request from the requesting agent is already in the buffer, the new request replaces the old request in the buffer.

If an incoming message is new information received from another agent, the secretary places the information into the theorem provers mailbox. The theorem prover incorporates the information in its current saturation level.

If an incoming message is a signal from the associated theorem prover, the secretary incorporates the new knowledge generated by the theorem prover and attempts to answer the requests in the buffer. For each message in the buffer the secretary constructs a reply from the agents database. If the reply is empty the request is left in the buffer, if the reply is not empty the reply is sent to the requesting agent.

4.3.3 Forward Progress Heuristic

In a distributed environment it is entirely possible that a given theorem proving agent may not have an appropriate combination of clauses in its database to infer a contradiction. Two unproductive situations could arise. One is a scenario in which no new clauses can be generated because the agent does not possess an appropriate combination of clauses to trigger its inference rule. In this situation, the agent has a clear indication it is not moving towards a contradiction. The second unproductive scenario is one in which the

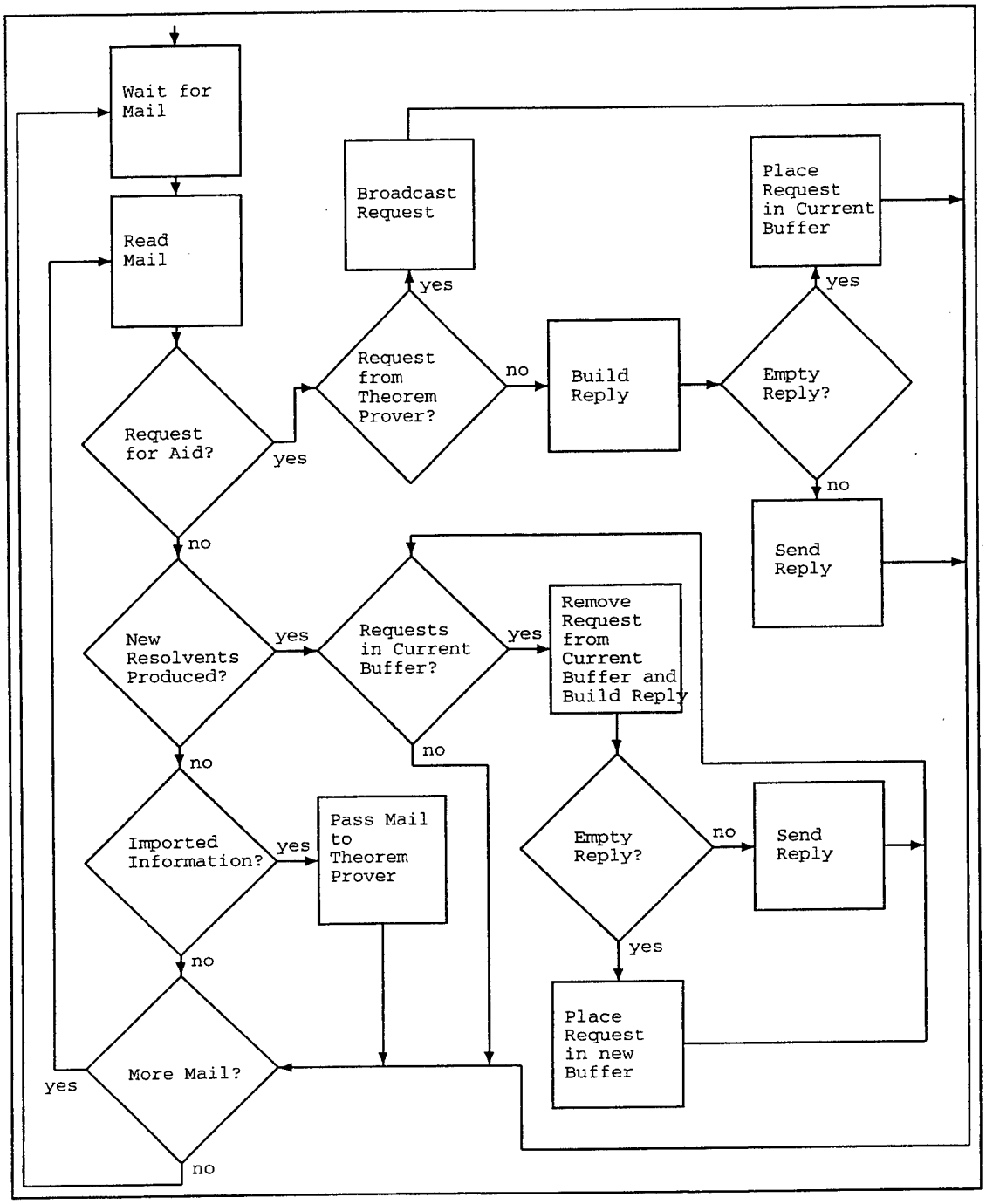


Figure 4.7: SHYRLI Secretary Control Structure.

agent can produce new clauses indefinitely but no effective progress is being made towards a solution. If this is the case, the agent must engage in introspection to assess how its work is progressing. This is accomplished through the use of heuristics. We have devised a heuristic for assessing progress in hyper-resolution based reasoning that is a generalization of the forward progress heuristic used in DARES.

SHYRLI performs hyper-resolution using a breadth first search strategy, where each saturation level corresponds to a level in the search tree. Recall the definition of a saturation level from the previous section. The forward progress heuristic is applied at the end of each saturation level. As was the case in DARES, our forward progress heuristic has two components: a proof advancing heuristic and a monotonic search heuristic. The proof advancing heuristic is used to detect whether an agent has advanced its proof towards a contradiction. It can only decide that the proof is advancing or that it is uncertain that the proof is advancing. In order to reduce the uncertainty we use the monotonic search heuristic.

The proof advancing heuristic examines the new resolvents in a saturation level to determine if they are moving towards a contradiction. If any new resolvent in a saturation level satisfies the proof advancing heuristic then the saturation level satisfies the proof advancing heuristic. As has been mentioned, the proof advancing heuristic used in SHYRLI is a generalization of the one used by DARES. Because DARES used binary resolution, its proof advancing heuristic assumed each clause would only have two parents. When using hyper-resolution a clause may have any number of parents. Our proof advancing heuristic is an extension of the proof advancing heuristic found in DARES that allows for this fact.

Key to the definition of the proof advancing heuristic is the concept of literal length. Literal length is defined as:

Definition: The *literal length* of a clause is the number of literals contained within that clause. In other terms: the literal length of a clause is the cardinality of the set representing that clause.

A clause R generated through hyper-resolution from clauses $\{E_1, \ldots, E_q, N\}$ satisfies the hyper-resolution proof advancing heuristic if:

1. Where c is the sum of the literal lengths of all the clauses in $\{E_1, \ldots, E_q, N\}$, the literal length r of R is less than $c - 2q$,
   or
2. R is a single literal,
   or
3. a member of $\{E_1, \ldots, E_q, N\}$ is a single literal.

A contradiction is detected when a resolvent of literal length zero is generated. In most cases resolution will produce a clause that has a longer literal length than its parents. A

resolvent that is shorter in literal length than its parents is an indication that the reasoning is moving towards a contradiction. When performing hyper-resolution, the maximum literal length of a resultant clause is $c - 2q$. If the literal length of a hyper-resolvent is less than this maximum the proof is seen as advancing.

Some resolvents, however, are desirable even though their literal length meets the upper bound. Under the unit preference strategy, resolution is conducted with unit clauses (clauses with one literal) whenever possible. Resolving with single literal clauses can be seen as advantageous because resolution of a single literal with a longer clause reduces the literal length of the longer clause by one. A resolvent generated using single literals as parents could still possibly meet the upper bound on clause length. Generation of a hyper-resolvent that has a single literal parent, or is a single literal itself is also seen as a sign that the proof is advancing.

The monotonic search heuristic is used when the proof advancing procedure fails to indicate forward progress. The monotonic search heuristic tries to determine if the theorem prover is narrowing its focus to fewer and fewer distinct predicate symbols. The monotonic search heuristic is defined as:

> Let α_n be the total number of distinct predicate symbols in the newly generated clauses at saturation level n. The search for the proof is said to be *monotonic* at level i if for $i > 1$, $\alpha_{i-1} > \alpha_i$.

With these two heuristics defined, the forward progress heuristic can be stated as follows:

> A proof is said to exhibit an apparent lack of forward progress at saturation level i if
>
> 1. The proof advancing heuristic is not satisfied at saturation level $i - 1$, *and*
> 2. The proof advancing heuristic is not satisfied at saturation level i, *and*
> 3. The search is not monotonic at saturation level i.

4.3.4 Formulation of Requests and Replies

What distinguishes this work from previous research is the cooperation strategies that are embodied in the strategies for formulation of requests for nonlocal aid and the responses to requests from other agents. The cooperation strategies used in DARES are strongly biased towards the unit preference strategy and set of support. Those that have been devised for SHYRLI take advantage of features found in hyper-resolution to focus attention more

quickly. The response strategy also permits agents to respond at a later time, should more information become available.

In any distributed problem solving environment we would like to limit the exchange of information between agents. The more information an agent receives, the greater the potential of its being less focused, and the greater the likelihood that more irrelevant and redundant inferences could be produced. The coordination strategy must limit the amount of information an agent will import, but still result in an exchange of information that will help problem solving progress.

Hyper-resolution as an inferencing mechanism divides the clauses into two subsets across an interpretation I: the set of *electrons* and the set of *nuclei*. To reduce amount of information exchanged, we restrict formulation of requests for information to one set and generation of responses to the other.

The issue of which set should be utilized to construct requests and which should be used in constructing responses is of some importance. In order to compute a resolvent there must exist a set S such that S contains one nucleus and q electrons where the largest predicates in the electrons relative to the ordering P unify with every negated atom in the nucleus. If an agent were to construct requests for new information using the electrons, it would expect to receive in response a set of nuclei with which it could potentially resolve. Suppose that $\{E_1, E_2, N_1\}$ and $\{E_3, E_4, N_2\}$ are each semantic clashes. Let A_1 and A_2 be agents. A_1 has clauses$\{N_1, E_1, E_3\}$ in its database and A_2 has clauses $\{N_2, E_2, E_4\}$ in its database. It is clear that neither A_1 sending the clause N_1 to A_2 nor A_2 sending the clause N_2 to A_1 would allow either agent to discover a semantic clash. It is possible, however, to exchange electrons in such a manner as to generate semantic clashes. If A_1 transmits E_3 to A_2 and A_2 transmits E_2 to A_1, two semantic clashes are possible and hyper-resolution can proceed.

We have chosen a strategy where the information that will be exchanged between agents consists of electrons. Remember that information exchanges are in response to requests for aid from a specific agent. In order for electrons to be useful to the requesting agent they must be able to participate in a semantic clash. The criteria for membership in a semantic clash is tied to the negated atoms in the nucleus that is participating in the semantic clash. Therefore, in responding to a request an agent must know what type of things the requesting agents needs.

The simplest strategy would be for an agent to formulate and transmit a request for information that contains the nuclei known to that agent. This would result in an inordinate amount of information exchange. To reduce the volume of inter-agent message traffic, we could form a set that is the union of all the literals in the nuclei. Remember that agents expect to receive sets of electrons in response to their requests for information. Electrons only contain positive literals. Thus we can further restrict requests so that they contain just the negated literals in the nuclei and still elicit the same information. We thus construct a request set that is the union of all the negated literals in the nuclei. To reduce the overhead

incurred in the responding agent we negate all the literals in the request set. Therefore the request set only consists of positive literals.

Now that we have formulated a mechanism for constructing requests for information, we must specify how responses should be constructed. As was mentioned earlier, responses should only be constructed using the set of electrons. The response strategy should only return electrons that resolve with literals in the request set. A nucleus can only resolve with an electron if the electron's largest predicate (relative to the ordering P) can resolve with one of the nucleus's negated literals. An electron should be considered as a candidate for response only if its largest predicate resolves with a member of the request set.

4.3.5 Privacy

The behavior engendered by this coordination strategy is different from that observed in most other automated reasoning systems. In other systems, an agent must be able to share all of its knowledge (in the worst case). There is no concept of private knowledge that is never shared. In our system the nuclei are never made known to other agents. Requests derived from the nuclei are, but the internal structure of an agent's nuclei is never seen by other agents. Furthermore, it is possible for a SHYRLI agent to keep a set of clauses private and still guarantee that the system will achieve the goal. To see this suppose that Q is a subset of the predicate symbols contained in the system. Consider the case in which all clauses containing predicate symbols in Q are initially known only to agent A. Suppose further that the members of Q are assigned the largest values in the ordering P. Other agents will never formulate a request containing a member of Q (because no predicate symbol in Q is in their sets of nuclei). Agent A will never respond to a request with an electron containing a member of Q (because the member of Q would be the largest predicate symbol in such an electron and therefore would never satisfy the membership criteria of the response set). Agent A need never place a member of Q in its request set (because no other agent has an electron containing a member of Q). Thus knowledge of a set of predicate symbols and all clauses containing those symbols can be kept private to an agent through the choice of the ordering P.

In section 4.3.1 we assumed a complete global ordering P. If there are private predicate symbols this does not have to be the case. As long as the set of globally known predicate symbols is completely ordered among the set of agents, each set of agents may have its own ordering which specifies the ordering of its private predicate symbols. The only stipulation on this ordering is that the private predicate symbols come before the global ones in the ordering P. There is no ordering relation necessary between two agents private predicate symbols. P only has to be a partial ordering over the entire set of predicate symbols known to the system.

There is one other mechanism in the request/response formulation strategy of SHYRLI which allows a SHYRLI agent to maintain information as private. SHYRLI agents can be

seen as "specialists." An agent never receives any new nuclei, nor does it share its own nuclei with the other agents. This means that if (in the proof of some hypothesis) a combination of electrons and a particular nucleus is necessary, one of the agents that has that nucleus in its database must participate in the proof.

4.3.6 A Three Agent SHYRLI Example

So that the reader will better understand how SHYRLI coordinates its activities to prove a theorem we will present a small example using three agents. The thereom we will use SHYRLI to prove is the one given as an example in the previous section. The example presented here was generated using SHYRLI. The nuclei were distributed randomly among the three agents. The clause distribution is shown in figure 4.8, figure 4.9 and figure 4.10.

electrons:		
AE1:	F(C)	(Axiom)
AE2:	G(D)	(Axiom)
AE3:	H(B) $\vee$ R(A)	(Axiom)
nuclei:		
AN1:	$\neg$J(A)	(Axiom)
AN2:	P(A) $\vee$ $\neg$G(x)	(Axiom)

Figure 4.8: Agent A's Clauses.

electrons:		
BE1:	F(C)	(Axiom)
BE2:	G(D)	(Axiom)
BE3:	H(B) $\vee$ R(A)	(Axiom)
nuclei:		
BN1:	I(y) $\vee$ $\neg$F(z) $\vee$ $\neg$P(x) $\vee$ $\neg$Q(x)	(Axiom)
BN2:	Q(E(x)) $\vee$ T(x) $\vee$ $\neg$R(x)	(Axiom)

Figure 4.9: Agent B's Clauses.

A mapping of these axiom designators to the deduction tree of the single agent example can be seen in Figure 4.11. This tree is identical to the deduction tree in the previous section, Figure 4.2 except for the names of the axioms. As before, circles represent given axioms and the negated theorem, boxes represent the derived clauses. The electrons that were derived earliest are higher in the tree than electrons that were derived later.

We are going to choose the same ordering as was chosen in the single agent example

electrons:		
CE1:	F(C)	(Axiom)
CE2:	G(D)	(Axiom)
CE3:	H(B) $\vee$ R(A)	(Axiom)
nuclei:		
CN1:	$\neg$P(x) $\vee$ $\neg$T(x)	(Axiom)
CN2:	P(y) $\vee$ $\neg$S(A,y)	(Axiom)
CN3:	J(x) $\vee$ $\neg$I(C)	(Axiom)
CN4:	$\neg$H(x)	(Axiom)
CN5:	S(x,E(x)) $\vee$ T(x) $\vee$ $\neg$R(x)	(Negated Theorem)

Figure 4.10: Agent C's Clauses.

of the previous section: alphabetical ordering by predicate symbol. Figure 4.12 shows the deduction tree for the proof and which agents derived which hyper-resolvents. Figure 4.13, Figure 4.14, and Figure 4.15 show the steps taken for each agent as they proceed with the problem.

Notice that no clauses are produced that were not produced in the one agent example in the previous section. This three agent proof follows the same path to the contradiction as the single agent example. This will be the case in all proofs where there is only one path to the goal. It is not necessarily the case when there are multiple paths.

We will describe in detail agent A's activities during the course of the proof. Agent A begins by assessing its current state. It decides that outside assistance is not needed as it has not tried to resolve its current electrons with its set of nuclei.

At time 2 agent A infers the hyper-resolvent **AHR1**, which was the result of the semantic clash {**AE2**, **AN2**}. Agent A now tries to compute a semantic clash using this new hyper-resolvent. This activity produces no new resolvents. At this point agent A has computed all the hyper-resolvents it can, it judges that outside assistance is now neccesary and constructs a request.

While agent A is constructing its request, a new request arrives from agent B. Agent A's secretary begins constructing a reply to this request. The predicate F(x) in the request set from agent B unifies with the largest predicate symbol in the clause **AE1**. The predicate P(x)in the request set from agent B unifies with the largest predicate symbol in **AHR1**. Thus **AE1** and **AHR1** are included in the reply to agent B.

At this point agent A has constructed its own requests consisting of the negated literals in its nuclei. This set is broadcast to the other two agents. At time 10 agent A receives replies from the other two agents. This reply contains no helpful information. Both of the clauses are duplicates of information agent A already contains. Agent A resends the

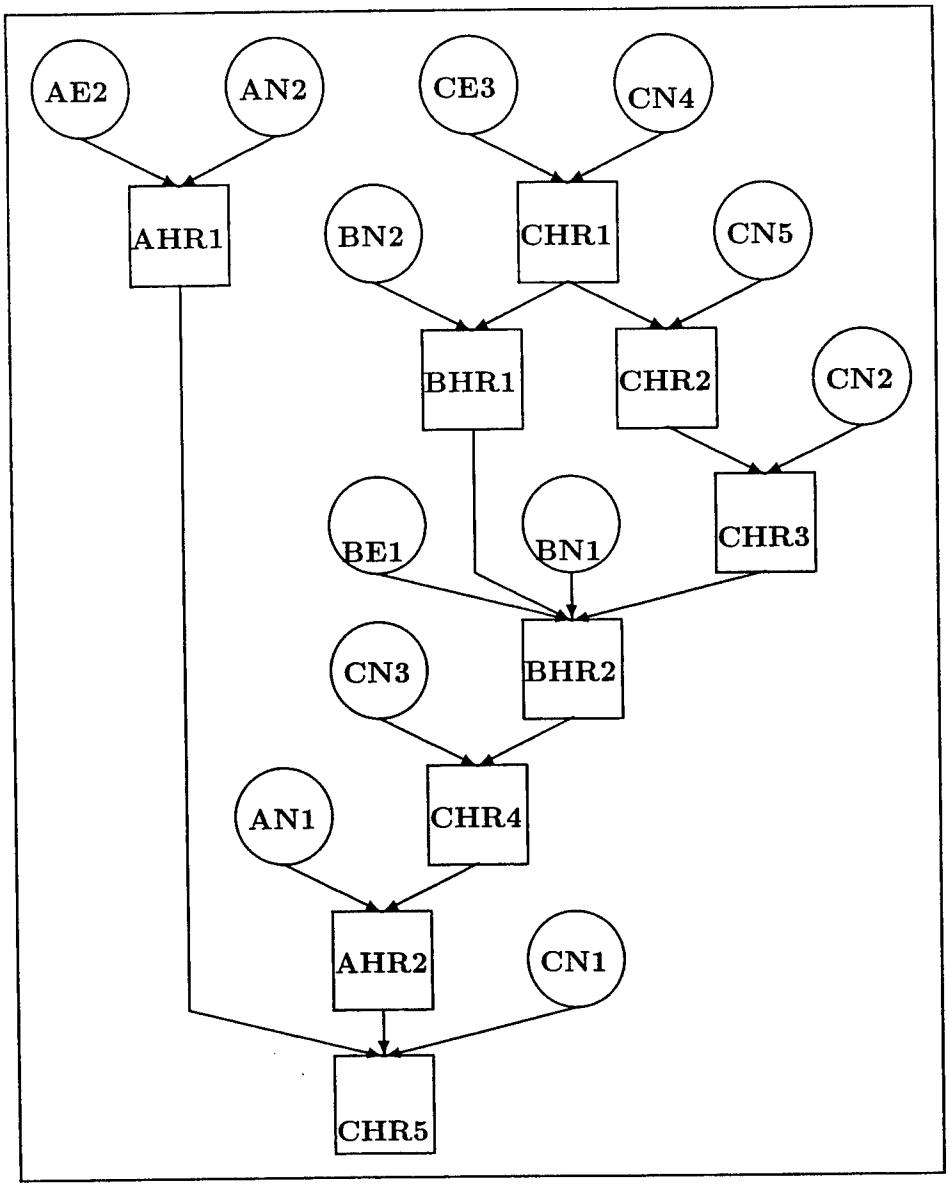


Figure 4.11: Deduction tree for the single agent example.

request.

At time 12 a request arrives from agent C. **AHR1** is considered as part of the reply since its largest predicate unifies with P(x) in agent C's request set. Similarly the largest predicate in **AE3** unifies with H(x) in agent C's request set. These two clauses, **AE3** and **AHR1**, are sent to agent C as the reply.

At time 21 another request arrives from agent C. As no more clauses have been inferred that might be of use to agent C, agent A enqueues the request.

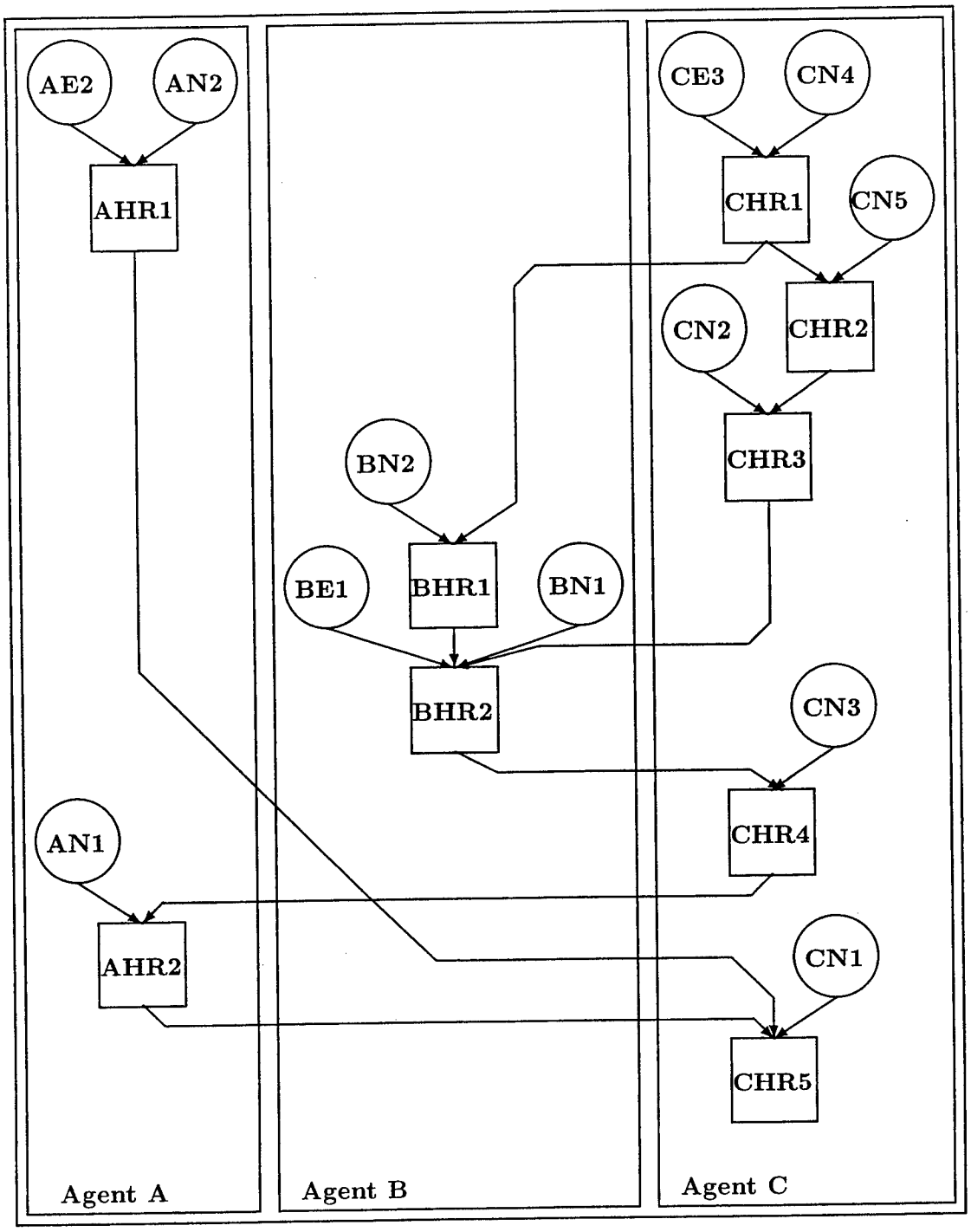


Figure 4.12: Deduction tree for the three agent example.

A clause, **CHR4** is imported at time 27 from agent C. Agent A then tries to perform an

time 1:		Determined Outside Assistance Unnecessary.
time 2:		Inference Step:
		AHR1: P(A) Parents: **AE2**, **AN2**.
time 3:		Determined Outside Assistance Unnecessary.
time 4:		Inference Step: no clauses produced.
time 5:		Determined Outside Assistance Necessary.
time 6:	(secretary)	Request from Agent B: {F(x), P(x), Q(x), R(x)}.
	(secretary)	Responding with: **AHR1**, **AE1**.
time 6:		Sending a request: {J(A), G(x)}.
		Waiting for replies.
time 10:		Imported Axioms:
		BE2: G(D).
		CE2: G(D).
time 11:		**BE2** deleted, equivalent to **AE2**.
time 11:		**CE2** deleted, equivalent to **AE2**.
time 12:	(secretary)	Request from Agent C: {P(x), T(x), S(A,y), I(C), H(x), R(x)}.
	(secretary)	Responding with: **AHR1**, **AE3**.
time 12:		Determined Outside Assistance Unnecessary.
time 13:		Sending a request: {J(A), G(x)}.
		Waiting for replies.
time 19:	(secretary)	Request from Agent B: {F(x), P(x), Q(x), R(x)}.
	(secretary)	No clauses for response, request placed on queue.
time 21:	(secretary)	Request from Agent C: {P(x), T(x), S(A,y), I(C), H(x), R(x)}.
	(secretary)	No clauses for response, request placed on queue.
time 27:		Imported Axioms:
		CHR4: J(x) $\vee$ T(A).
time 28:		Inference Step:
		ARH2: T(A) Parents: **CHR4**, **AN1**.
time 28:	(secretary)	Responding to enqueued Agent C's request with : **ARH2**.
time 29:		Determined Outside Assistance Unnecessary.
time 30:		Inference Step: no clauses produced.
time 31:	(secretary)	Request from Agent C: {P(x), T(x), S(A,y), I(C), H(x), R(x)}.
	(secretary)	No clauses for response, request placed on queue.
time 31:		Sending a request: {J(A), G(x)}.
		Waiting for replies.

Figure 4.13: Agent A's Steps

inference step using this clause. The hyper-resolvent **ARH2** is computed from the semantic clash {**CHR4**, **AN1**}. Agent A then determines that outside assistance is not necessary and attempts another inference step.

At time 28 agent A's secretary responds to the enqueued request of agent C with the clause **ARH2**.

Agents A's next inference step produces no new resolvents. Agent A then computs that outside assistance is necessary and broadcasts a request.

time 1:		Determined Outside Assistance Unnecessary.
time 2:		Inference Step: no clauses produced.
time 3:		Determined Outside Assistance Necessary.
time 4:		Sending a request: {F(x), P(x), Q(x), R(x)}.
		Waiting for replies.
time 8:	(secretary)	Request from Agent A: {J(A), G(x)}.
	(secretary)	Responding with: **BE2**.
time 8:		Imported Axioms:
		CHR3: P(E(A)) ∨ T(A)
		CHR1: R(A)
		CE1: F(C).
		AHR1: P(A).
		AE1: F(C).
time 9:		**CE1** deleted, equivalent to **AE1**.
time 9:		**AE1** deleted, equivalent to **AE1**.
time 10:		Determined Outside Assistance Unnecessary.
time 11:		Inference Step:
		BHR1: Q(E(A)) ∨ T(A) Parents: **CHR1**, **BN2**.
time 12:	(secretary)	Request from Agent C: {P(x), T(x), S(A,y), I(C), H(x), R(x)}.
	(secretary)	No clauses for response, request placed on queue.
time 12:		Determined Outside Assistance Unnecessary.
time 13:		Inference Step:
		BHR2: I(x) ∨ T(A) Parents: **CHR3**, **BE1**, **BHR1**, **BN1**.
time 14:		Determined Outside Assistance Unnecessary.
time 15:	(secretary)	Request from Agent A: {J(A), G(x)}.
	(secretary)	No clauses for response, request placed on queue.
time 15:		Inference Step: no clauses produced.
time 16:		Determined Outside Assistance Necessary.
time 17:		Sending a request: {F(x), P(x), Q(x), R(x)}.
		Waiting for replies.
time 21:	(secretary)	Request from Agent C: {P(x), T(x), S(A,y), I(C), H(x), R(x)}.
	(secretary)	Responding with: **BHR2**, **BE3**.
time 33:	(secretary)	Request from Agent A: {J(A), G(x)}.
	(secretary)	No clauses for response, request placed on queue.

Figure 4.14: Agent B's Steps.

At time 31 a request arrives from agent C. This request is enqueued as there are no clauses that fit the requirements for the response set in agent A's database of clauses.

Agent A now halts waiting for mail. It is agent C that eventually proves the theorem. It is not necessarily the case that all agents in SHYRLI will solve the problem, as long as the forward progress heuristic is an accurate judge of progress it is the case that at least one of them will find a contradiction if one exists.

time 1:		Determined Outside Assistance Unnecessary.
time 2:		Inference Step:
		CHR1: R(A) Parents: **CE3**, **CN4**.
time 3:		Determined Outside Assistance Unnecessary.
time 4:		Inference Step:
		CHR2: S(A,E(A)) $\vee$ T(A) Parents: **CHR1**, **CN5**.
time 5:		Determined Outside Assistance Unnecessary.
time 6:		Inference Step:
		CHR3: P(E(A)) $\vee$ T(A) Parents: **CHR2**, **CN2**.
time 6:	(secretary)	Request from Agent B: {F(x), P(x), Q(x), R(x)}.
	(secretary)	Responding with: **CHR3**, **CHR1**, **CE1**.
time 7:		Determined Outside Assistance Unnecessary.
time 8:	(secretary)	Request from Agent A: {J(A), G(x)}.
	(secretary)	Responding with: **CE2**.
time 8:		Inference Step: no clauses produced.
time 9:		Determined Outside Assistance Necessary.
time 10:		Sending a request: {P(x), T(x), S(A,y), I(C), H(x), R(x)}.
		Waiting for replies.
time 14:		Imported Axioms:
		AHR1: P(A).
		AE3: H(B) $\vee$ R(A).
time 15:		**AE3** deleted, equivalent to **CE3**.
time 15:	(secretary)	Request from Agent A: {J(A), G(x)}.
	(secretary)	No clauses for response, request placed on queue.

Figure 4.15: Agent C's Steps, Part 1.

4.4 Experimentation

The goal of this research is to study domain independent issues in distributed automated reasoning. To facilitate this we have developed a distributed automated reasoning system that is not tied to any domain. SHYRLI's use of the predicate calculus and hyper-resolution meets this domain independence requirement.

We have implemented SHYRLI. SHYRLI is written in Common LISP and runs on TI Explorers under SIMULACT, a distributed testbed environment developed at Clarkson University [61] [54] [56].

Our initial experiments with SHYRLI demonstrate its ability to perform distributed reasoning using randomly selected distributions of knowledge (clauses). The agents in SHYRLI have generated significantly fewer irrelevant clauses in the production of a proof than did the agents in DARES with the same knowledge distributions. This is to be expected, given the use of hyper-resolution by SHYRLI agents.

In this section we will show how SHYRLI can be applied to a specific domain, the simulation of digital circuits. We then present a set of experiments that has allowed us

time 16:		Inference Step: no clauses produced.
time 17:		Determined Outside Assistance Necessary.
time 19:	(secretary)	Request from Agent B: {F(x), P(x), Q(x), R(x)}.
	(secretary)	No clauses for response, request placed on queue.
time 19:		Sending a request: {P(x), T(x), S(A,y), I(C), H(x), R(x)}.
		Waiting for replies.
time 23:		Imported Axioms:
		BHR2: I(x) ∨ T(A).
		BE3: H(B) ∨ R(A).
time 24:		**BE3** deleted, equivalent to **CE3**.
time 25:		Inference Step:
		CHR4: J($A3664$) ∨ T(A) Parents: **BHR2**, **CN3**.
time 25:	(secretary)	Responding to enqueued Agent A's request with: **CHR4**.
time 26:		Determined Outside Assistance Unnecessary.
time 27:		Inference Step: no clauses produced.
time 28:		Determined Outside Assistance Necessary.
time 29:		Sending a request: {P(x), T(x), S(A,y), I(C), H(x), R(x)}.
		Waiting for replies.
time 30:		Imported Axioms:
		ARH2: T(A).
time 31:	(secretary)	Request from Agent A: {J(A), G(x)}.
	(secretary)	No clauses for response, request placed on queue.
time 32:		Inference Step:
time 33:		**CHR5:** □ Parents: **AHR1**, **AHR2**, **CN1**.
time 34:		Contradiction detected: Done Processing.

Figure 4.16: Agent C's Steps, Part 2.

to gain some insight into SHYRLI's behavior. Finally we explain the difference between SHYRLI's search space and a single agents search space.

4.4.1 A Domain Specific Example.

To show that through the predicate calculus SHYRLI can reason about specific domains, we have implemented a small domain specific example. This example is a simple simulator for digital circuits. It allows the abstraction of a complex device made from simpler elements. An agent using a complex device does not have to know how the internal parts of the device work. When trying to simulate a circuit containing a complex device, an agent can ask other agents who understand the device in question to simulate the components of the device for it [1].

Digital circuits are made up of devices and connections. Our simulator can only represent simple combinatorial logic made from these objects, aside from this limitation, it is

[1]The inspiration for this example came from [30]

quite robust. We have chosen to represent the devices as constants and the connections as predicates.

A device is a constant in our representation. The following predicate associates a specific type with a constant:

Type(*Scope*,*Device*,*Type*).

The symbols in italics are variables. This predicate associates the type *Type* with the device *Device* in the scope *Scope*. The types that can be used in the *Type* field consist of but are not limited to the following list:

And-Gate - Represents a logical AND gate.

Or-Gate - Represents a logical OR gate.

Xor-Gate - Represents a logical XOR gate.

Not-Gate - Represents a logical NOT gate.

This list can be extended through a mechanism which allows the creation of new devices consisting of existing devices.

The *Scope* field defines the name space or scope in which the device is defined. Every device in a scope must have a unique name. Scope provides a way of dividing the problem between agents. An agent can be assigned a specific scope to work in. This assignment of scope can be used to divide a complex problem into logical parts that can be distributed over a group of agents. The mechanism that allows the abstraction of complex devices include methods that pass values between scopes. The use of this scoping mechanism is not tied to the domain of digital circuits.

Each device also has pins or links to other devices. A devices links are characterized by the Link function. The format of the Link function is:

Link(*Link-Type*,*Pin*,*Device*).

where *Pin* is a specific pin on the device *Device*. Different *Link-Type*'s may have the same numbered pin on the same device. The values of *Link-Type* are limited to the following constants:

Input - Input to a simple gate.

Output - Output from a simple gate.

Down - Input to a complex device.

Up - Output from a complex device.

NOT gates have links:

Link(Output,P1,*Not-Gate*)

Link(Input,P1,*Not-Gate*)

All other simple gates have links:

Link(Output,P1,*Two-Input-Gate*)

Link(Input,P2,*Two-Input-Gate*)

Link(Input,P1,*Two-Input-Gate*)

In order to connect devices together the predicate Connect is used. The format of the Connect predicate is:

Connect(*Scope*,*Link1*,*Link2*).

where *Scope* is the scope in which the connection is valid. *Link1* and *Link2* are links and *Link1* is connected to *Link2*. It is important to make the connections flow in a forward direction (i.e. from outputs to inputs).

The following is an example of Connect:

Type(Scope-A,N1,Not-Gate).

Type(Scope-A,A1,And-Gate).

Connect(Scope-A,Link(Output,P1,A1),Link(Input,P2,Not-Gate)).

This example says that we have two devices, N1 and A1. N1 is a constant representing a NOT gate in scope Scope-A. A1 is a constant representing an AND gate in scope Scope-A. The Connect predicate states that in the scope Scope-A the pin one output of the AND gate, A1, is connected to the pin two input of the NOT gate, N1.

To place a value on a specific pin the Value predicate is used. The value predicate had the following format:

Value(*Scope*,*Link*, *Value*,*Return*).

Scope is the scope in which we are working. *Link* is the pin with which we are associating *Value* (which may be either the constant High or Low). *Return* is used by the sentences of the simulator to indicate where the outputs of a complex device should be sent. *Return* is used as a stack. If it becomes necessary to simulate a complex device the simulator "shifts" into the scope in which the device is defined in order to simulate it. In order to remember

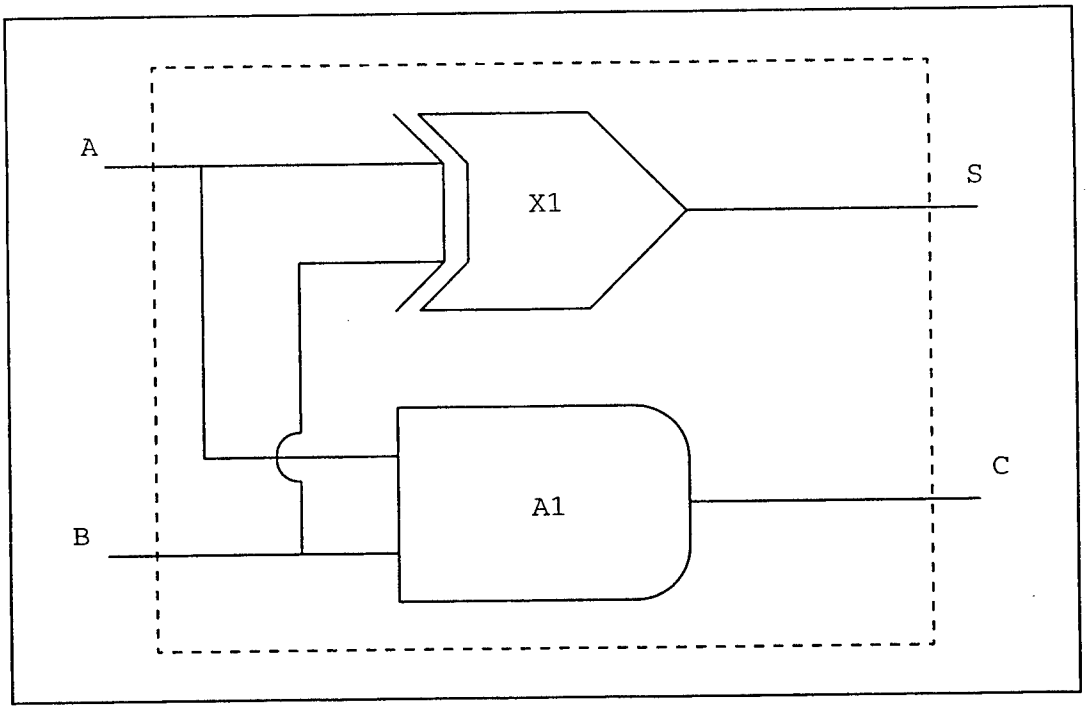


Figure 4.17: Circuit Diagram of a Half-Adder.

where the outputs of the complex device should be sent the *Return* stack is used. When providing values to the simulator the constant value "nil" should be used in the *Return* field.

As an example, suppose we want to create a new device called a Half-Adder which simulates the functionality of a half-adder. The circuit diagram for a half-adder is shown in Figure 4.17. We will need two gates, an XOR gate and an AND gate:

Type(Scope-B,X1,Xor-Gate).

Type(Scope-B,A1,And-Gate).

We are creating this device in the scope Scope-B. Within a scope all device names must be unique, the same device name may be used to represent different devices in different scopes.

We are creating an abstract type Half-Adder in the scope Scope-B so we add the following predicate.

Understands(Half-Adder,Scope-B).

If a device of type Half-Adder is used, the simulator will know that the definition of a Half-Adder is in the scope Scope-B.

The following predicates are used to represent the connections in a Half-Adder:

Connect(Scope-B,Link(Input,P1,Half-Adder),Link(Input,P1,X1)).

Connect(Scope-B,Link(Input,P2,Half-Adder),Link(Input,P2,X1)).

Connect(Scope-B,Link(Input,P1,Half-Adder),Link(Input,P1,A1)).

Connect(Scope-B,Link(Input,P2,Half-Adder),Link(Input,P2,A1)).

Connect(Scope-B,Link(Output,P1,X1),Link(Up,P1,Half-Adder)).

Connect(Scope-B,Link(Output,P1,A1),Link(Up,P2,Half-Adder)).

The final two Connect predicates specify that the outputs of X1 and A1 are used as the outputs of the complex device Half-Adder.

We will now present the predicate calculus sentences that implement the simulator. The following sentences describe how connections work:

C1: ($\forall$ *Scope*, *Pin*, *Device*, *Link*, *Value*, *Return*)
(Connect(*Scope*, Link(Input, *Pin*, *Device*), *Link*) $\wedge$
Value(*Scope*, Link(Input, *Pin*, *Device*), *Value*, *Return*)) $\rightarrow$
Value(*Scope*, *Link*, *Value*, *Return*).

C2: ($\forall$ *Scope*, *Pin*, *Device*, *Link*, *Value*, *Return*)
(Connect(*Scope*, Link(Output,*Pin*, *Device*), *Link*) $\wedge$
Value(*Scope*, Link(Output, *Pin*, *Device*), *Value*, *Return*)) $\rightarrow$
Value(*Scope*, *Link*, *Value*, *Return*).

C3: ($\forall$ *Scope-1*, *Pin*, *Device-1*, *Value*, *Scope-2*, *Device-2*, *Return*)
Value(*Scope-1*, Link(Up, *Pin*, *Device-1*), *Value*, R(*Scope-2*, *Device-2*, *Return*)) $\rightarrow$
Value(*Scope-2*, Link(Output, *Pin*, *Device-2*), *Value*, *Return*).

C4: ($\forall$ *Scope-1*, *Pin*, *Device*, *Value*, *Return*, *Type*, *Scope-2*)
(Value(*Scope-1*, Link(Down, *Pin*, *Device*), *Value*, *Return*) $\wedge$
Type(*Scope-1*, *Device*, *Type*) $\wedge$
Understands(*Type*, *Scope-2*)) $\rightarrow$
Value(*Scope-2*, Link(Input, *Pin*, *Type*), *Value*, R(*Scope-1*, *Device*, *Return*)).

C1 and **C2** are sentences which describe the behavior of input and output links. The sentences simply state that if a value of a link, *link-1*, is known and the link is connected to another link, *link-2*, then *link-2* has the same value as *link-1*.

The two final sentences are more complex and deal with complex devices. In order for the sentences describing a device to be used with more than one Value predicate the return

field is used. The R function is used as a stack to hold the scope and name of the device that is representing the instantiation of a complex device. Sentence **C4** states that if a value, *Value*, is known for a down link, and *Scope-2* has the definition of the device in the down link then the input link of the device has *Value* for a value. The return field *Return* holds the original name of the device to differentiate this value from any others that might be using the same circuit definition. *Return* also holds the scope of the original name of the device.

The definition of **C3** states what to do when an up link has a value. This sentence uses the values in the R function to give the orginal device name instantiated values for its outputs. The R function also provides a mechanism to return to the scope in which the complex device was imbedded.

The following predicate calculus sentences describe the simple gate elements:

AND1: (∀ *Scope*, *device*, *Return*)
(Type(*Scope*, *Device*, And-Gate) ∧
Value(*Scope*, Link(Input, P1, *Device*), High, *Return*) ∧
Value(*Scope*, Link(Input, P2, *Device*), High, *Return*)) →
Value(*Scope*, Link(Output, P1, *Device*), High, *Return*).

AND2: (∀ *Scope*, *Device*, *Pin*, *Return*)
(Type(*Scope*, *Device*, And-Gate) ∧
Value(*Scope*, Link(Input, *Pin*, *Device*), Low, *Return*)) →
Value(*Scope*, Link(Output, P1, *Device*), Low, *Return*).

OR1: (∀ *Scope*, *device*, *Return*)
(Type(*Scope*, *Device*, Or-Gate) ∧
Value(*Scope*, Link(Input, P1, *Device*), Low, *Return*) ∧
Value(*Scope*, Link(Input, P2, *Device*), Low, *Return*)) →
Value(*Scope*, Link(Output, P1, *Device*), Low, *Return*).

OR2: (∀ *Scope*, *Device*, *Pin*, *Return*)
(Type(*Scope*, *Device*, Or-Gate) ∧
Value(*Scope*, Link(Input, *Pin*, *Device*), High, *Return*)) →
Value(*Scope*, Link(Output, P1, *Device*), High, *Return*).

XOR1: (∀ *Scope*, *device*, *Value*, *Return*)
(Type(*Scope*, *Device*, Xor-Gate) ∧
Value(*Scope*, Link(Input, P1, *Device*), *Value*, *Return*) ∧
Value(*Scope*, Link(Input, P2, *Device*), *Value*, *Return*)) →
Value(*Scope*, Link(Output, P1, *Device*), Low, *Return*).

XOR2: (∀ *Scope*, *device*, *Return*)
(Type(*Scope*, *Device*, Xor-Gate) ∧

Value(*Scope*, Link(Input, P1, *Device*), Low, *Return*) ∧
Value(*Scope*, Link(Input, P2, *Device*), High, *Return*)) →
Value(*Scope*, Link(Output, P1, *Device*), High, *Return*).

XOR3: (∀ *Scope*, *device*, *Return*)
(Type(*Scope*, *Device*, Xor-Gate) ∧
Value(*Scope*, Link(Input, P1, *Device*), High, *Return*) ∧
Value(*Scope*, Link(Input, P2, *Device*), Low, *Return*)) →
Value(*Scope*, Link(Output, P1, *Device*), High, *Return*).

NOT1: (∀ *Scope*, *Device*, *Pin*, *Return*)
(Type(*Scope*, *Device*, Not-Gate) ∧
Value(*Scope*, Link(Input, P1, *Device*), Low, *Return*)) →
Value(*Scope*, Link(Output, P1, *Device*), High, *Return*).

NOT2: (∀ *Scope*, *Device*, *Return*)
(Type(*Scope*, *Device*, Not-Gate) ∧
Value(*Scope*, Link(Input, P1, *Device*), High, *Return*)) →
Value(*Scope*, Link(Output, P1, *Device*), Low, *Return*).

In order for these sentences to be used in SHYRLI they must be converted into Skolem normal form. The resulting sentences are shown in Figure 4.18.

We will now show a complete example of how SHYRLI can reason about knowledge expressed in this domain. Our example will have two scopes and two agents, Agent A and Agent B. We will explicitly assign an agent to each domain. To accomplish this each agent will receive a copy of the sentences in Figure 4.18 with the variable *Scope* replaced with the constant that represents the agents name. In this manner agents will only be able to reason about things within their scope.

We will be simulating a full adder (see Figure 4.19). Agent A has the definition of a full adder in it scope. The full-adder consists of two half-adders and an OR gate. Agent A however does not have the definition for a half-adder in its scope. Agent B has the definition of the half adder contained within its scope.

Besides the modified clauses in Figure 4.18, Agent A has the clauses contained in Figure 4.20. Agent B also has modified clauses from Figure 4.18 and the clauses contained in Figure 4.21.

Agent A's clauses **S1–S3** in Figure 4.20 supply initial input values to the full adder. The clause **NH1** is the negated hypothesis.

When SHYRLI starts processing agent B attempts to perform an inference step it fails. Realizing that outside assistance is necessary agent B sends a request to agent A. At this time agent A cannot handle the request, so agent A enqueues the request for a later saturation level.

C1:	¬Connect(*Scope*, Link(Input, *Pin*, *Device*), *Link*) ∨ ¬Value(*Scope*, Link(Input, *Pin*, *Device*), *Value*, *Return*) ∨ Value(*Scope*, *Link*, *Value*, *Return*).
C2:	¬Connect(*Scope*, Link(Output,*Pin*, *Device*), *Link*) ∨ ¬Value(*Scope*, Link(Output, *Pin*, *Device*), *Value*, *Return*) ∨ Value(*Scope*, *Link*, *Value*, *Return*).
C3:	¬Value(*Scope*, Link(Up, *Pin*, *Device-1*), *Value*, R(*Scope-2*, *Device-2*, *Return*) ∨ Value(*Scope-2*, Link(Output, *Pin*, *Device-2*), *Value*, *Return*).
C4:	¬Value(*Scope*, Link(Down, *Pin*, *Device*), *Value*, *Return*) ∨ ¬Type(*Scope*, *Device*, *Type*) ∨ ¬Understands(*Type*, *Scope-2*)) ∨ Value(*Scope-2*, Link(Input, *Pin*, *Type*), *Value*, R(*Scope*, *Device*, *Return*)).
AND1:	¬Type(*Scope*, *Device*, And-Gate) ∨ ¬Value(*Scope*, Link(Input, P1, *Device*), High, *Return*) ∨ ¬Value(*Scope*, Link(Input, P2, *Device*), High, *Return*) ∨ Value(*Scope*, Link(Output, P1, *Device*), High, *Return*).
AND2:	¬Type(*Scope*, *Device*, And-Gate) ∨ ¬Value(*Scope*, Link(Input, *Pin*, *Device*), Low, *Return*) ∨ Value(*Scope*, Link(Output, P1, *Device*), Low, *Return*).
OR1:	¬Type(*Scope*, *Device*, Or-Gate) ∨ ¬Value(*Scope*, Link(Input, P1, *Device*), Low, *Return*) ∨ ¬Value(*Scope*, Link(Input, P2, *Device*), Low, *Return*) ∨ Value(*Scope*, Link(Output, P1, *Device*), Low, *Return*).
OR2:	¬Type(*Scope*, *Device*, Or-Gate) ∨ ¬Value(*Scope*, Link(Input, *Pin*, *Device*), High, *Return*) ∨ Value(*Scope*, Link(Output, P1, *Device*), High, *Return*).
XOR1:	¬Type(*Scope*, *Device*, Xor-Gate) ∨ ¬Value(*Scope*, Link(Input, P1, *Device*), *Value*, *Return*) ∨ ¬Value(*Scope*, Link(Input, P2, *Device*), *Value*, *Return*) ∨ Value(*Scope*, Link(Output, P1, *Device*), Low, *Return*).
XOR2:	¬Type(*Scope*, *Device*, Xor-Gate) ∨ ¬Value(*Scope*, Link(Input, P1, *Device*), Low, *Return*) ∨ ¬Value(*Scope*, Link(Input, P2, *Device*), High, *Return*) ∨ Value(*Scope*, Link(Output, P1, *Device*), High, *Return*).
XOR3:	¬Type(*Scope*, *Device*, Xor-Gate) ∨ ¬Value(*Scope*, Link(Input, P1, *Device*), High, *Return*) ∨ ¬Value(*Scope*, Link(Input, P2, *Device*), Low, *Return*) ∨ Value(*Scope*, Link(Output, P1, *Device*), High, *Return*).
NOT1:	¬Type(*Scope*, *Device*, Not-Gate) ∨ ¬Value(*Scope*, Link(Input, P1, *Device*), Low, *Return*) ∨ Value(*Scope*, Link(Output, P1, *Device*), High, *Return*).
NOT2:	¬Type(*Scope*, *Device*, Not-Gate) ∨ ¬Value(*Scope*, Link(Input, P1, *Device*), High, *Return*) ∨ Value(*Scope*, Link(Output, P1, *Device*), Low, *Return*).

Figure 4.18: Skolem normal form sentences for digital circuit simulation.

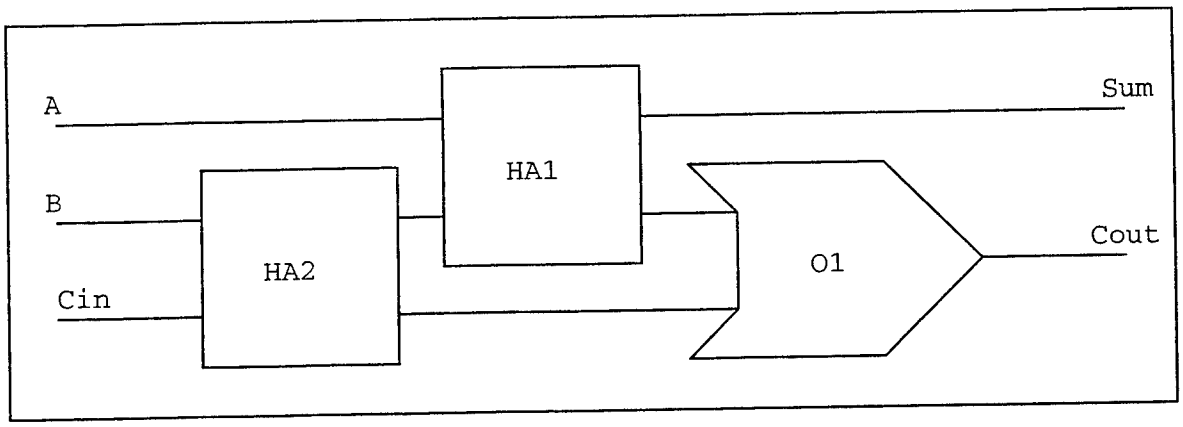


Figure 4.19: Circuit Diagram of a Full Adder.

A1:	Type(Agent-A,F1,Full-Adder).
A2:	Type(Agent-A,O1,Or-Gate).
A3:	Type(Agent-A,HA1,Half-Adder).
A4:	Type(Agent-A,HA2,Half-Adder).
A5:	Connect(Agent-A,Link(Input,A,F1),Link(Down,P1,HA1)).
A6:	Connect(Agent-A,Link(Input,B,F1),Link(Down,P1,HA2)).
A7:	Connect(Agent-A,Link(Input,Cin,F1),Link(Down,P2,HA2)).
A8:	Connect(Agent-A,Link(Output,P1,HA1),Link(Output,Sum,F1)).
A9:	Connect(Agent-A,Link(Output,P2,HA1),Link(Input,P1,O1)).
A10:	Connect(Agent-A,Link(Output,P1,HA2),Link(Down,P2,HA1)).
A11:	Connect(Agent-A,Link(Output,P2,HA2),Link(Input,P2,O2)).
A12:	Connect(Agent-A,Link(Output,P1,O1),Link(Output,Cout,F1)).
S1:	Value(Agent-A,Link(Input,A,F1),High,Nil).
S2:	Value(Agent-A,Link(Input,B,F1),High,Nil).
S3:	Value(Agent-A,Link(Input,Cin,F1),Low,Nil).
NH1:	¬Value(Agent-A,Link(Output,Cout,F1),High,Nil).

Figure 4.20: Circuit Example, Agent A Clauses.

Agent A is able to infer some hyper-resolvents:

HRA1: Value(Agent-A,Link(Down,P1,HA1),High,Nil).
Parents: **S1**, **A5**, **C1**.

HRA2: Value(Agent-A,Link(Down,P1,HA2),High,Nil).
Parents: **S2**, **A6**, **C1**.

HRA3: Value(Agent-A,Link(Down,P2,HA2),Low,Nil).
Parents: **S3**, **A7**, **C1**.

B1:	Understands(Half-Adder,Agent-B).
B2:	Type(Agent-B,X1,Xor-Gate).
B3:	Type(Agent-B,A1,And-Gate).
B4:	Connect(Agent-B,Link(Input,P1,Half-Adder),Link(Input,P1,X1)).
B5:	Connect(Agent-B,Link(Input,P2,Half-Adder),Link(Input,P2,X1)).
B6:	Connect(Agent-B,Link(Input,P1,Half-Adder),Link(Input,P1,A1)).
B7:	Connect(Agent-B,Link(Input,P2,Half-Adder),Link(Input,P2,A1)).
B8:	Connect(Agent-B,Link(Output,P1,X1),Link(Up,P1,Half-Adder)).
B9:	Connect(Agent-B,Link(Output,P1,A1),Link(Up,P2,Half-Adder)).

Figure 4.21: Circuit Example, Agent B Clauses.

At this point agent A attempts another inference step and fails. Agent A formulates a request and sends it to agent B. Agent B responds with **B1**. Agent A can now continue inferencing and infers the following clauses:

HRA4: Value(Agent-B,Link(Input,P1,Half-Adder),High,R(Agent-A,HA1,Nil)).
Parents: **HRA1**, **A3**, **B1**, **C4**.

HRA5: Value(Agent-B,Link(Input,P1,Half-Adder),High,R(Agent-A,HA2,Nil)).
Parents: **HRA2**, **A4**, **B1**, **C4**.

HRA6: Value(Agent-B,Link(Input,P2,Half-Adder),Low,R(Agent-A,HA2,Nil)).
Parents: **HRA3**, **A4**, **B1**, **C4**.

Agent A realizes that these clauses (**HRA4**, **HRA5**, and **HRA6**) fit the requirements of agent B's request. Agent A sends these clauses to agent B. After attempting another inference step and failing to produce any new hyper-resolvents agent A sends a request and waits.

After receiving the clauses (**HRA4**, **HRA5**, and **HRA6**) from agent A, agent B produces the following hyper-resolvents:

HRB1: Value(Agent-B,Link(Input,P1,X1),High,R(Agent-A,HA2,Nil)).
Parents: **B4**, **HRA5**, **C1**.

HRB2: Value(Agent-B,Link(Input,P1,X1),High,R(Agent-A,HA1,Nil)).
Parents: **B4**, **HRA4**, **C1**.

HRB3: Value(Agent-B,Link(Input,P2,X1),Low,R(Agent-A,HA2,Nil)).
Parents: **B5**, **HRA6**, **C1**.

HRB4: Value(Agent-B,Link(Input,P1,A1),High,R(Agent-A,HA2,Nil)).
Parents: **B6**, **HRA5**, **C1**.

HRB5: Value(Agent-B,Link(Input,P1,A1),High,R(Agent-A,HA1,Nil)).
Parents: **B6**, **HRA4**, **C1**.

HRB6: Value(Agent-B,Link(Input,P2,A1),Low,R(Agent-A,HA2,Nil)).
Parents: **B7**, **HRA6**, **C1**.

After this inference step agent B can infer the following hyper-resovents:

HRB7: Value(Agent-B,Link(Output,P1,A1),Low,R(Agent-A,HA2,Nil)).
Parents: **B3**, **HRB6**, **AND2**.

HRB8: Value(Agent-B,Link(Output,P1,X1),High,R(Agent-A,HA2,Nil)).
Parents: **B2**, **HRB1**, **HRB3**, **XOR3**.

After this inference step agent B can infer the following hyper-resolvents:

HRB9: Value(Agent-B,Link(Up,P1,Half-Adder),High,R(Agent-A,HA2,Nil)).
Parents: **B8**, **HRB8**, **C2**.

HRB10: Value(Agent-B,Link(Up,P2,Half-Adder),Low,R(Agent-A,HA2,Nil)).
Parents: **B9**, **HRB7**, **C2**.

After this inference step agent B can infer the following hyper-resolvents:

HRB11: Value(Agent-A,Link(Output,P1,HA2),High,Nil).
Parents: **HRB9**, **C3**.

HRB12: Value(Agent-A,Link(Output,P2,HA2),Low,Nil).
Parents: **HRB10**, **C3**.

Agent B realizes that the clauses **HRB9** and **HRB10** match the requirements of agent A's request. Agent B sends these clauses to agent A. After attempting another inference step and failing to produce any new hyper-resolvents agent B sends a request and waits.

After receiving the clauses **HRB9** and **HRB10** from agent B, agent A produces the following hyper-resolvents:

HRA7: Value(Agent-A,Link(Down,P2,HA1),High,Nil).
Parents: **A10**, **HRB11**, **C2**.

HRA8: Value(Agent-A,Link(Input,P2,O2),Low,Nil).
Parents: **A11**, **HRB12**, **C2**.

After this inference step agent A can infer the following hyper-resolvent:

HRA9: Value(Agent-B,Link(Input,P2,Half-Adder),High,R(Agent-A,HA1,Nil)).
Parents: **B1**, **A3**,**HRA7**, **C4**.

Agent A realizes that the clause **HRA9** fits the requirements of agent B's request. Agent A sends this clause to agent B. After attempting another inference step and failing to produce any new hyper-resolvents agent A sends a request and waits.

After receiving the clause **HRA9** from agent A, agent B produces the following hyper-resolvents:

HRB13: Value(Agent-B,Link(Input,P2,X1),High,R(Agent-A,HA1,Nil)).
Parents: **HRA9**, **B5**, **C1**.

HRB14: Value(Agent-B,Link(Input,P2,A1),High,R(Agent-A,HA1,Nil)).
Parents: **HRA9**, **B7**, **C1**.

After this inference step agent B can infer the following hyper-resolvents:

HRB15: Value(Agent-B,Link(Output,P1,X1),Low,R(Agent-A,HA1,Nil)).
Parents: **B2**, **HRB2**, **HRB13**, **XOR1**.

HRB16: Value(Agent-B,Link(Output,P1,A1),High,R(Agent-A,HA1,Nil)).
Parents: **B3**, **HRB5**, **HRB14**, **AND1**.

After this inference step agent B can infer the following hyper-resolvents:

HRB17: Value(Agent-B,Link(Up,P1,Half-Adder),Low,R(Agent-A,HA1,Nil)).
Parents: **HRB15**, **B8**, **C2**.

HRB18: Value(Agent-B,Link(Up,P2,Half-Adder),High,R(Agent-A,HA1,Nil)).
Parents: **HRB16**, **B9**, **C2**.

After this inference step agent B can infer the following hyper-resolvents:

HRB19: Value(Agent-A,Link(Output,P1,HA1),Low,Nil).
Parents: **HRB17**, **C3**.

HRB20: Value(Agent-A,Link(Output,P2,HA1),High,Nil).
Parents: **HRB18**, **C3**.

Agent B realizes that the clauses **HRB19** and **HRB20** fit the requirements of agent A's request. Agent B sends this clause to agent A. After attempting another inference step and failing to produce any new hyper-resolvents agent B sends a request and waits.

After receiving the clauses **HRB19** and **HRB20** from agent B, agent A produces the following hyper-resolvents:

HRA10: Value(Agent-A,Link(Output,Sum,F1),Low,Nil).
Parents: **A8**, **HRB19**, **C2**.

HRA11: Value(Agent-A,Link(Input,P1,O1),High,Nil).
Parents: **A9**, **HRB20**, **C2**.

After this inference step agent A can infer the following hyper-resolvent:

HRA12: Value(Agent-A,Link(Output,P1,O1),High,Nil).
Parents: **A2**, **HRA11**, **OR2**.

After this inference step agent A can infer the following hyper-resolvent:

HRA13: Value(Agent-A,Link(Output,Cout,F1),High,Nil).
Parents: **A12**, **HRA12**, **C2**.

After this inference step agent A can infer the following hyper-resolvent:

HRA14: □.
Parents: **HRA13**, **NH1**.

At this point the theorem is proved and our hypothesis is proven correct. Notice that by looking at the clauses produced we can find the values of all the pins in the digital circuit.

SHYRLI's base mechanism has no understanding of digital circuits, however the knowledge given SHYRLI allows SHYRLI to reason about different domains. SHYRLI has no knowledge about specific domains except what is given it in terms of the predicate calculus. Even then, SHYRLI applies one interpretation to the clauses, the one that is necessary for hyper-resolution.

4.4.2 Impact of Different Knowledge Distributions

In DARES it was noted that certain distributions had adverse affects on performance, however even in its worse case the distributed cases outperformed the single agent case significantly. This behavior is does not seem to be the appear in SHYRLI. In SHYRLI, hyper-resolution prunes the search space extensively so any pruning as a result of a distribution of knowledge is minimized.

This pruning of the search space leads to fewer paths through the search space to the goal. In DARES there are potentially more paths to follow to a contradiction than in SHYRLI. This means that the allowable paths to a contradiction might wind their way through knowledge held by several different agents in SHYRLI, while in DARES a path might be found that uses fewer agents or less coordination. The winding of a path through the agents is distribution dependent, and would indicate that SHYRLI is very sensitive to specific distributions of knowledge.

A1:	$I(y) \vee \neg F(z) \vee \neg P(x) \vee \neg Q(x)$	(Axiom)
A2:	$\neg P(x) \vee \neg T(x)$	(Axiom)
A3:	$P(y) \vee \neg S(A,y)$	(Axiom)
A4:	$J(x) \vee \neg I(C)$	(Axiom)
A5:	$\neg J(A)$	(Axiom)
A6:	$\neg H(x)$	(Axiom)
A7:	$F(C)$	(Axiom)
A8:	$G(D)$	(Axiom)
A9:	$H(B) \vee R(A)$	(Axiom)
A10:	$P(A) \vee \neg G(x)$	(Axiom)
A11:	$Q(E(x)) \vee T(x) \vee \neg R(x)$	(Axiom)
A12:	$S(x,E(x)) \vee T(x) \vee \neg R(x)$	(Negated Theorem)

Figure 4.22: Clauses for the coordination experiment.

4.4.2.1 Sensitivity to Distributions

As an example of two different knowledge distributions that have vastly different processing times on SHYRLI we present two different distributions of axioms for two agents from the example in section 4.2.8. The axioms we are using are shown in Figure 4.22. The two distributions are shown in Figure 4.23. The resulting times and broadcasts for assistance for the two different distributions are shown in Figure 4.24.

Distribution One	
Agent A	**A3, A6, A7, A8, A9, A11, A12.**
Agent B	**A1, A2, A4, A5, A7, A8, A9, A10.**
Distribution Two	
Agent A	**A2, A3, A4, A6, A7, A8, A9, A10**
Agent B	**A1, A5, A7, A8, A9, A11, A12.**

Figure 4.23: Distributions in the coordination experiment.

	Distribution One	Distribution Two
Processing Time	11 seconds	33 seconds
Number of Broadcasts	3	8

Figure 4.24: Results of the coordination experiment.

The reason for the increase in processing time in the second distribution is the increase in coordination needed to traverse the search space to a contradiction. To get a graphical picture of this, Figure 4.25 shows the deduction tree of the first distribution. Notice how that only two coordination points are necessary. To compute the hyper-resolvent **A18** Agent B must import two clauses from Agent A. In Figure 4.26 notice that there are many coordination points. In order for an Agent to generate a hyper-resolvent it was almost always the case that the agent had to import knowledge. This behavior leads to a higher coordination cost for SHYRLI.

4.4.2.2 Experiments

In order to judge the impact that different distributions have on SHYRLI's performance we have devised a random distribution experiment. The experimental evidence given in this section supports our view that different distributions can have signicant effects on the performance of SHYRLI.

To minimize the behavior characteristic of a particular set of clauses we have run our experiment on three different sets of clauses. The test sets can be seen in Figure 4.22, Figure 4.27 and Figure 4.30. We have run fifty random distributions of each of these sets of clauses on a SHYRLI network of three agents. The experimental results show that different distributions have vastly different communication patterns.

We have assumed that there is no overlap (or redundancy) of information in the SHYRLI network of agents. At the start of theorem-proving no agent possesses information that another agent has. We devised an automatic collecting system that collected the following information during a run:

- Processing time.
- Time agents spent waiting for replies from other agents.
- Number of requests an agent broadcast.
- Size of an agents requests.
- Number of replies an agent received.
- Number of electrons an agent produced.
- Number of electrons an agent received.

Each experiment was performed on a cold-booted lisp machine to reduce the effect of garbage collection on the processing time parameter. The clauses used in experiment 1 are the same clauses that were used in the general behavior experiment in [56].

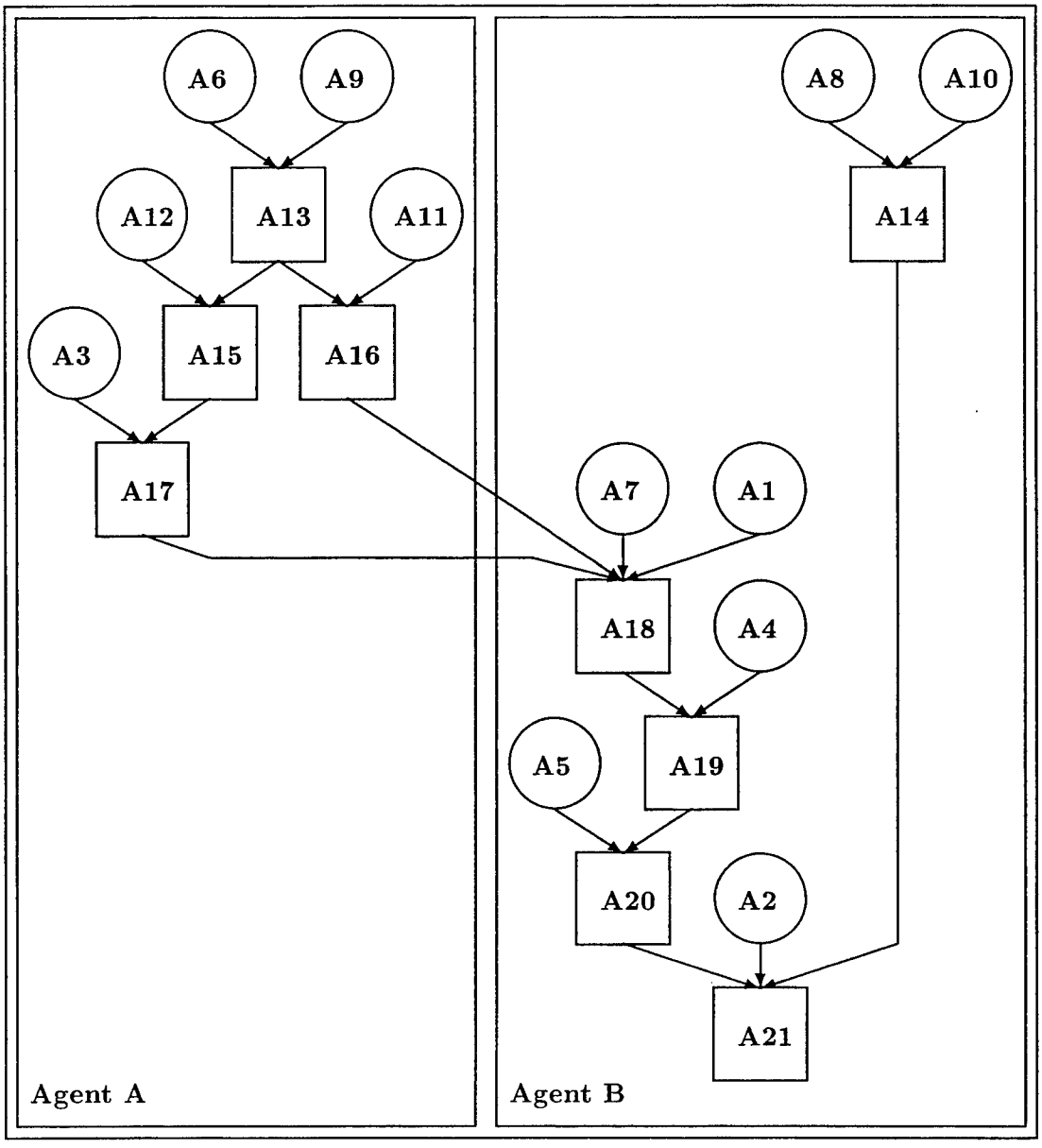


Figure 4.25: Deduction tree for distribution one.

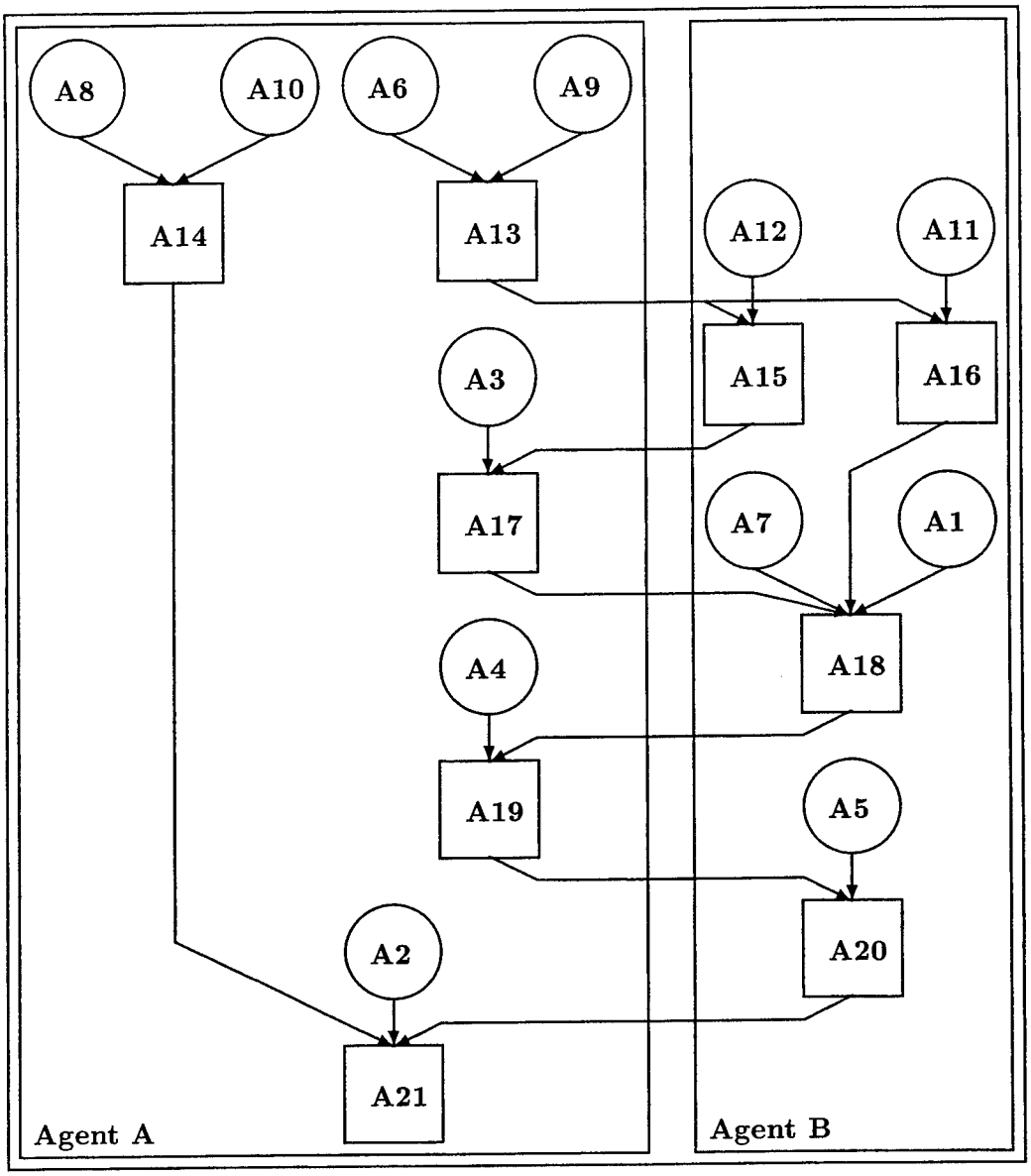


Figure 4.26: Deduction tree for distribution two.

In our experiments we have seen that different distributions exhibit markedly different performances. As a general rule, it seems that as the number of broadcasts for assistance increases, the time to solve the problem increases also. Data from each of these three sets can be ssen in the graphs in Figure 4.32, Figure 4.33, and Figure 4.34. These graphs plot the number of broadcasts for assistance versus the time to find a contradiction.

The variance of the data is high. These graphs tell us that the number of broadcasts for assistance is not as closely tied to the amount of time necessary to solve the problem as

A1:	¬Value(Actor-A,Link(Output,Cout,F1),High,C).
A2:	Value(Actor-A,Link(Input,D,F1),High,Nil).
A3:	Value(Actor-A,Link(Input,C,F1),Low,Nil).
A4:	Value(Actor-A,Link(Input,B,F1),High,Nil).
A5:	Value(Actor-A,Link(Input,A,F1),High,Nil).
A6:	Connect(Actor-B,Link(Output,P1,A1),Link(Up,P2,Half-Adder)).
A7:	Connect(Actor-B,Link(Output,P1,X1),Link(Up,P1,Half-Adder)).
A8:	Connect(Actor-B,Link(Input,P2,Half-Adder),Link(Input,P2,A1)).
A9:	Connect(Actor-B,Link(Input,P1,Half-Adder),Link(Input,P1,A1)).
A10:	Connect(Actor-B,Link(Input,P2,Half-Adder),Link(Input,P2,X1)).
A11:	Connect(Actor-B,Link(Input,P1,Half-Adder),Link(Input,P1,X1)).
A12:	Type(Actor-B,A1,And-Gate).
A13:	Type(Actor-B,X1,Xor-Gate).
A14:	Understands(Half-Adder,Actor-B).
A15:	Connect(Actor-A,Link(Output,P1,N2),Link(Input,Cin,F1)).
A16:	Connect(Actor-A,Link(Output,P1,A2),Link(Input,P1,N2)).
A17:	Connect(Actor-A,Link(Output,P1,N1),Link(Input,P2,A2)).
A18:	Connect(Actor-A,Link(Output,P1,A1),Link(Input,P1,N1)).
A19:	Connect(Actor-A,Link(Output,P1,O2),Link(Input,P1,A2)).
A20:	Connect(Actor-A,Link(Input,D,F1),Link(Input,P2,A1)).
A21:	Connect(Actor-A,Link(Input,C,F1),Link(Input,P1,A1)).
A22:	Connect(Actor-A,Link(Input,D,F1),Link(Input,P2,O2)).
A23:	Connect(Actor-A,Link(Input,C,F1),Link(Input,P1,O2)).
A24:	Connect(Actor-A,Link(Output,P1,O1),Link(Output,Cout,F1)).
A25:	Connect(Actor-A,Link(Output,P2,Ha2),Link(Input,P2,O1)).

Figure 4.27: Clauses for the second Random Distribution Experiment (part 1 of 3).

we thought it would be. This is because the number of broadcasts for assistance is not as close a measure of the amount of coordination necessary to solve the problem as we had at first believed.

We surmised that a better way to measure the amount of coordination necessary would be to measure the wait time of all agents. As this was not one of the parameters we originally decided to measure we ran the test cases again on fifty distributions of test set one and twenty of test set three. Our assumption seems to have been correct. A great deal of the agents time seems to be spent waiting for other agents to derive results or in time lost in communicating over the network. The graphs of this data can be seen in Figure 4.35 and Figure 4.36.

In order to reduce the time the agents spend waiting for mail we reduced the cost of communication to see how the system performs. The results (shown in Figure 4.37 we obtained indicate that the system performance is still very linear, only the slope seems to have shifted. In this example, unlike the earlier trials with this data set, the best multiple agent case outperformed the single agent case.

A26:	Connect(Actor-A,Link(Output,P1,Ha2),Link(Down,P2,Ha1)).
A27:	Connect(Actor-A,Link(Output,P2,Ha1),Link(Input,P1,O1)).
A28:	Connect(Actor-A,Link(Output,P1,Ha1),Link(Output,Sum,F1)).
A29:	Connect(Actor-A,Link(Input,Cin,F1),Link(Down,P2,Ha2)).
A30:	Connect(Actor-A,Link(Input,B,F1),Link(Down,P1,Ha2)).
A31:	Connect(Actor-A,Link(Input,A,F1),Link(Down,P1,Ha1)).
A32:	Type(Actor-A,N2,Not-Gate).
A33:	Type(Actor-A,N1,Not-Gate).
A34:	Type(Actor-A,A2,And-Gate).
A35:	Type(Actor-A,A1,And-Gate).
A36:	Type(Actor-A,O2,Or-Gate).
A37:	Type(Actor-A,F1,Full-Adder).
A38:	Type(Actor-A,O1,Or-Gate).
A39:	Type(Actor-A,Ha2,Half-Adder).
A40:	Type(Actor-A,Ha1,Half-Adder).
A41:	Value(*Scope*,Link(Output,P1,*Device*),Low, *Return*) ∨ ¬Type(*Scope*,*Device*,Not-Gate) ∨ ¬Value(*Scope*,Link(Input,P1,*Device*),High, *Return*).
A42:	Value(*Scope*,Link(Output,P1,*Device*),High, *Return*) ∨ ¬Type(*Scope*,*Device*,Not-Gate) ∨ ¬Value(*Scope*,Link(Input,P1,*Device*),Low, *Return*).
A43:	Value(*Scope*,Link(Output,P1,*Device*),High, *Return*) ∨ ¬Type(*Scope*,*Device*,Xor-Gate) ∨ ¬Value(*Scope*,Link(Input,P2,*Device*),High, *Return*) ∨ ¬Value(*Scope*,Link(Input,P1,*Device*),Low, *Return*).
A44:	Value(*Scope*,Link(Output,P1,*Device*),High, *Return*) ∨ ¬Type(*Scope*,*Device*,Xor-Gate) ∨ ¬Value(*Scope*,Link(Input,P2,*Device*),Low, *Return*) ∨ ¬Value(*Scope*,Link(Input,P1,*Device*),High, *Return*).

Figure 4.28: Clauses for the second Random Distribution Experiment (part 2 of 3).

These experiments confirmed our assumptions that SHYRLI would be very sensitive to distributions of knowledge. In cases where the deduction path to the contradiction winds itself through many agents, agents spend a great deal of time waiting for results from other agents.

When we examined the deduction trees of individual examples we were able to see that more coordination was necessary in distributions that the system took longer to prove than in distributions that the system proved quickly. We also noticed that in some faster cases deductions that could be done in parallel were distributed over the agents and not in a single agent.

We have been able to characterize the distributions along two dimensions, the amount of coordination necessary and the amount of parallelism. A graphical picture of this can be seen in Figure 4.38. The dimensions are tied together. In most cases where there is an

A45:	Value(*Scope*,Link(Output,P1,*Device*),Low, *Return*) ∨ ¬Type(*Scope*,*Device*,Xor-Gate) ∨ ¬Value(*Scope*,Link(Input,P2,*Device*),*V*, *Return*) ∨ ¬Value(*Scope*,Link(Input,P1,*Device*),*V*, *Return*).
A46:	Value(*Scope*,Link(Output,P1,*Device*),High, *Return*) ∨ ¬Type(*Scope*,*Device*,Or-Gate) ∨ ¬Value(*Scope*,Link(Input,*Pin*,*Device*), High,*Return*).
A47:	Value(*Scope*,Link(Output,P1,*Device*),Low, *Return*) ∨ ¬Type(*Scope*,*Device*,Or-Gate) ∨ ¬Value(*Scope*,Link(Input,P2,*Device*),Low, *Return*) ∨ ¬Value(*Scope*,Link(Input,P1,*Device*),Low, *Return*).
A48:	Value(*Scope*,Link(Output,P1,*Device*),Low, *Return*) ∨ ¬Type(*Scope*,*Device*,And-Gate) ∨ ¬Value(*Scope*,Link(Input,*Pin*,*Device*), Low,*Return*).
A49:	Value(*Scope*,Link(Output,P1,*Device*),High, *Return*) ∨ ¬Type(*Scope*,*Device*,And-Gate) ∨ ¬Value(*Scope*,Link(Input,P2,*Device*),High, *Return*) ∨ ¬Value(*Scope*,Link(Input,P1,*Device*),High, *Return*).
A50:	Value(*Actor*,Link(Input,*Pin*,*Type*), *Value*,R(*Scope*,*Device*,*Return*)) ∨ ¬Type(*Scope*,*Device*,*Type*) ∨ ¬Understands(*Type*,*Actor*) ∨ ¬Value(*Scope*,Link(Down,*Pin*,*Device*), *Value*,*Return*).
A51:	Value(*Actor*,Link(Output,*Pin*,*Tag*), *Value*,*Return*) ∨ ¬Value(*Scope*,Link(Up,*Pin*,*Device*), *Value*,R(*Actor*,*Tag*,*Return*)).
A52:	Value(*Scope*,*Link*,*Value*,*Return*) ∨ ¬Connect(*Scope*,Link(Output,*Pin*,*Device*), *Link*) ∨ ¬Value(*Scope*,Link(Output,*Pin*,*Device*), *Value*,*Return*).
A53:	Value(*Scope*,*Link*,*Value*,*Return*) ∨ ¬Connect(*Scope*,Link(Input,*Pin*,*Device*), *Link*) ∨ ¬Value(*Scope*,Link(Input,*Pin*,*Device*), *Value*,*Return*).

Figure 4.29: Clauses for the second Random Distribution Experiment (part 3 of 3).

attempt to increase parallelization, the amount of coordination necessary is increased. This works the other way as well. In attempts to decrease the amount of coordination necessary some parallelization is lost.

Figure 4.26 shows a distribution that is poor in parallelization and has a high amount of necessary coordination. The distribution in Figure 4.25 is better in parallelization and has a small amount of necessary coordination. Figure 4.12 shows a distribution that maximizes the potential parallelization of the axiom set, but increases the amount of necessary coordination over the distribution in Figure 4.25.

This insight into the different types of distributions has allowed us to understand why wait time is a good indicator of performance. High wait times mean that:

- Time was spent waiting for communication to travel between agents.

A1:	¬Equal(A,T4).
A2:	¬Equal(T4,T1).
A3:	¬Equal(T4,B).
A4:	¬Equal(A,T1).
A5:	¬Equal(A,B).
A6:	¬Equal(B,T1).
A7:	¬Equal(B,D).
A8:	¬Equal(D,C).
A9:	¬Equal(D,A).
A10:	¬Equal(B,C).
A11:	¬Equal(B,A).
A12:	¬Equal(A,C).
A13:	On(v,vp,G(wp,z)) ∨ ¬Clear(v,G(wp,z)) ∨ ¬On(v,vp,z) ∨ ¬On(u,v,z).
A14:	On(v,vp,G(wp,z)) ∨ ¬On(v,vp,z) ∨ ¬On(u,v,G(wp,z)).
A15:	Clear(vp,G(wp,z)) ∨ ¬On(u,vp,z) ∨ ¬State(F(A(u,v),w),G(wp, z)).
A16:	On(u,v,G(wp,z)) ∨ ¬State(F(A(u,v),w),G(wp, z)).
A17:	Clear(u,G(wp,z)) ∨ ¬State(F(A(u,v),w),G(wp, z)).
A18:	State(w,z) ∨ ¬State(F(y,w),z).
A19:	State(wp,G(wp,z)) ∨ ¬Context(x,wp,z) ∨ ¬Maxrobot(x).
A20:	¬Conflict(S(O),y,End,z).
A21:	Context(O,End,z).

Figure 4.30: Clauses for the Third Random Distribution Experiment (part 1 of 2).

- Time was spent waiting for other agents to derive a result.

The first case is dependent on the communication network on the system. As was shown in Figure 4.37, reducing this time can improve performance.

The second case is an aggregate of two phenomena: the amount of coordination necessary and the amount of parallelization in the problem. If the deduction tree is highly linear, then only one agent may be working on a piece at a time. In distributing this problem over a set of agents processing time will invariably suffer. Certain distributions could lead to a large amount of necessary coordination which impedes the performance of the system. If the deduction tree is highly parallel, the best case scenario leads to low wait time as the agents cooperate to accomplish parts of the problem at the same time. Most of the wait time in this case would come from the cost of communication. A side affect to parallelization is that the amount of necessary coordination might increase thereby increasing coordination cost.

We do not intend for these characterizations to be used by system architects in distributing axioms among a set of agents, but rather as a way to understand how a set of distributed agents behave. In the problems in which we are interested, the distribution of knowledge is not a variable we can control. We would like to be able to understand why our systems exhibit certain behaviors and SHYRLI has shown us that the distribution of

A22:	Conflict(x,y,w,z) ∨ Legal(x,y,w,z) ∨ ¬Clear(y,z) ∨ ¬Environment(y,x,z).
A23:	Conflict(x,y,w,z) ∨ Equal(y,v) ∨ Equal(y,u) ∨ ¬Conflict(S(x),y,F(A(u,v), w),z) ∨ ¬Context(x,F(A(u,v),w),z).
A24:	Context(x,F(A(u,v),w),z) ∨ Equal(u,v) ∨ ¬Block(u) ∨ ¬Legal(x,v,w,z) ∨ ¬Legal(x,u,w,z).
A25:	Environment(u,x,z) ∨ ¬Environment(v,x,z) ∨ ¬On(u,v,z).
A26:	Maxrobot(S(S(O))).
A27:	Environment(T4,S(S(O)),z).
A28:	Environment(T3,S(S(O)),z).
A29:	Environment(T2,S(S(O)),z).
A30:	Environment(T3,S(O),z).
A31:	Environment(T2,S(O),z).
A32:	Environment(T1,S(O),z).
A33:	On(A,T1,End).
A34:	On(B,T4,End).
A35:	On(D,T3,End).
A36:	On(C,T2,End).
A37:	Clear(A,End).
A38:	Clear(B,End).
A39:	Clear(D,End).
A40:	Clear(C,End).
A41:	Block(A).
A42:	Block(B).
A43:	Block(D).
A44:	Block(C).
A45:	¬On(B,T1,z) ∨ ¬On(A,T4,z).

Figure 4.31: Clauses for the Third Random Distribution Experiment (part 2 of 2).

knowledge can greatly affect system behavior.

We have found that for small problems the distributed case performs rather poorly in comparison to the single agent case. For a larger problem the best case begins to approach and even surpass the time for a single agent. Our first two set cases fit into the low parallelization category. Their distributions therefore run from low coordination to high coordination. Those distributions which maximize parallelization while minimizing coordination cost achieve the best performance and outperform the single agent case.

Test set three provides more parallelization. This means that a set of distributed agents has a better chance of outperforming the single agent case. Distributions of test set three can fall into any of the quadrants of Figure 4.38. Using a network with a high cost of communication, forty percent of the distributed cases outperformed the single agent case.

These experiments have not shown us the performance benefit of distributing the problem over a set of agents that was obtained using DARES. DARES reported a performance increase of about a factor of two in processing time. Although the authors were not looking

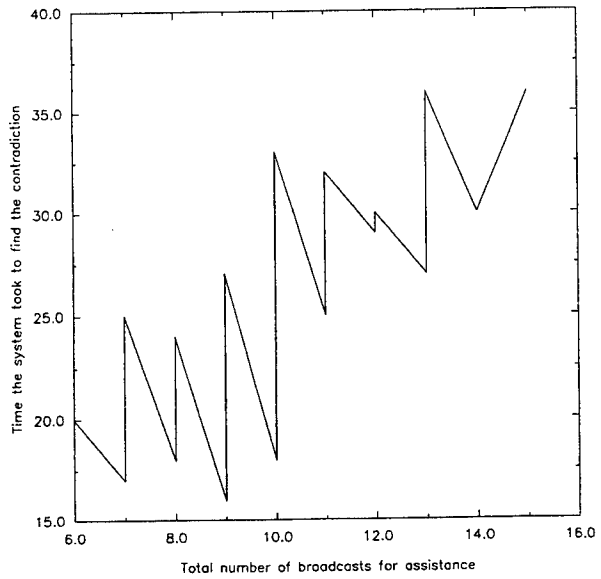


Figure 4.32: Test Set 1: Completion time vs. number of broadcasts.

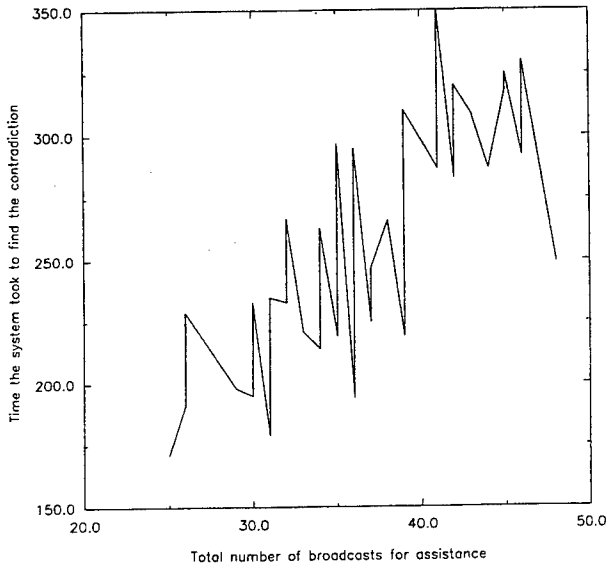


Figure 4.33: Test Set 2: Completion time vs. number of broadcasts.

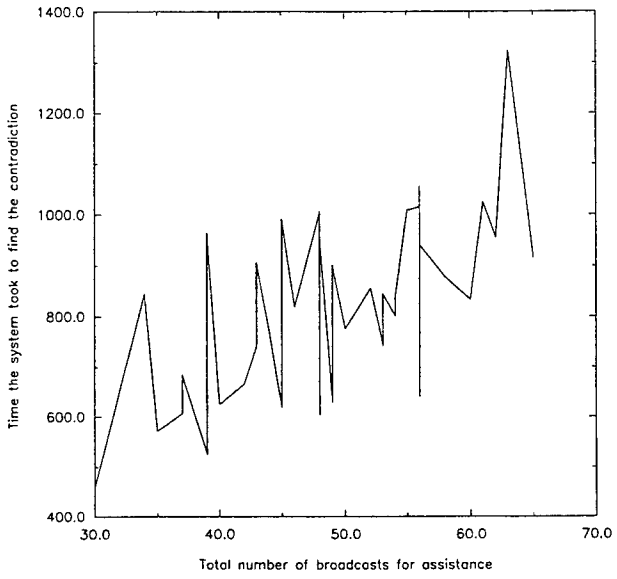


Figure 4.34: Test Set 3: Completion time vs. number of broadcasts.

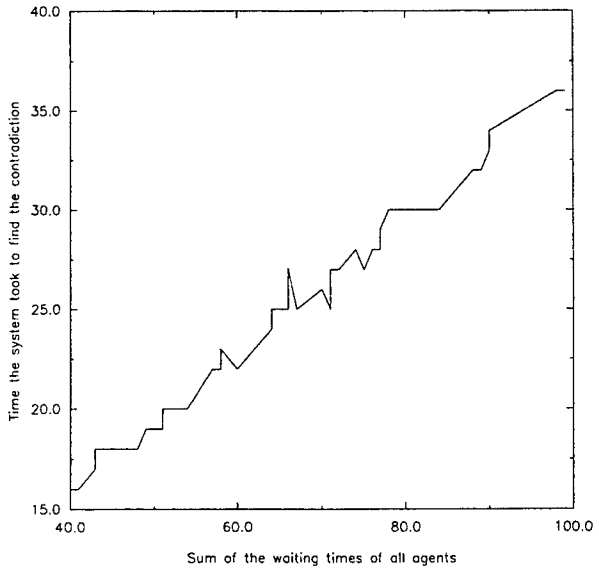


Figure 4.35: Test Set 1: Completion time vs. wait time.

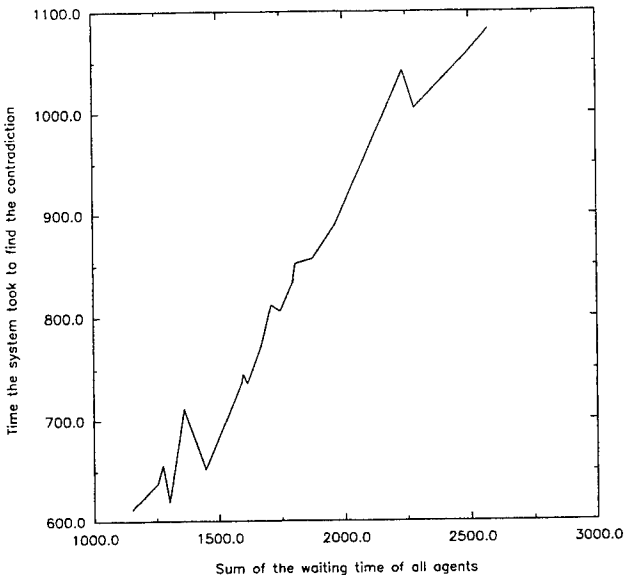


Figure 4.36: Test Set 3: Completion time vs. wait time.

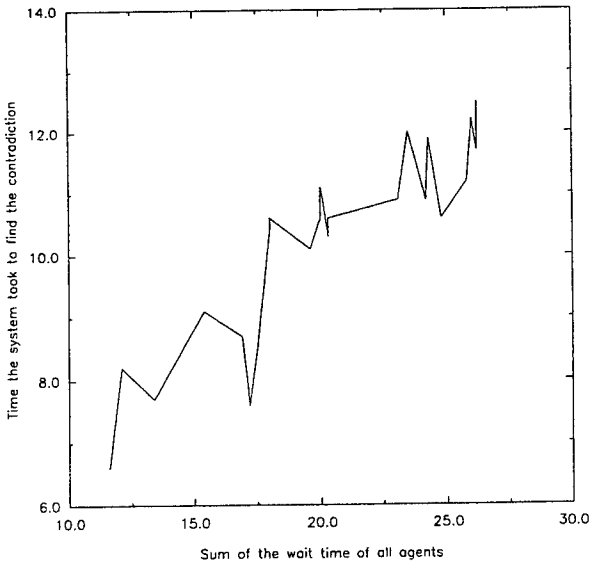


Figure 4.37: Test Set 1: Completion time vs. wait time with smaller communication cost.

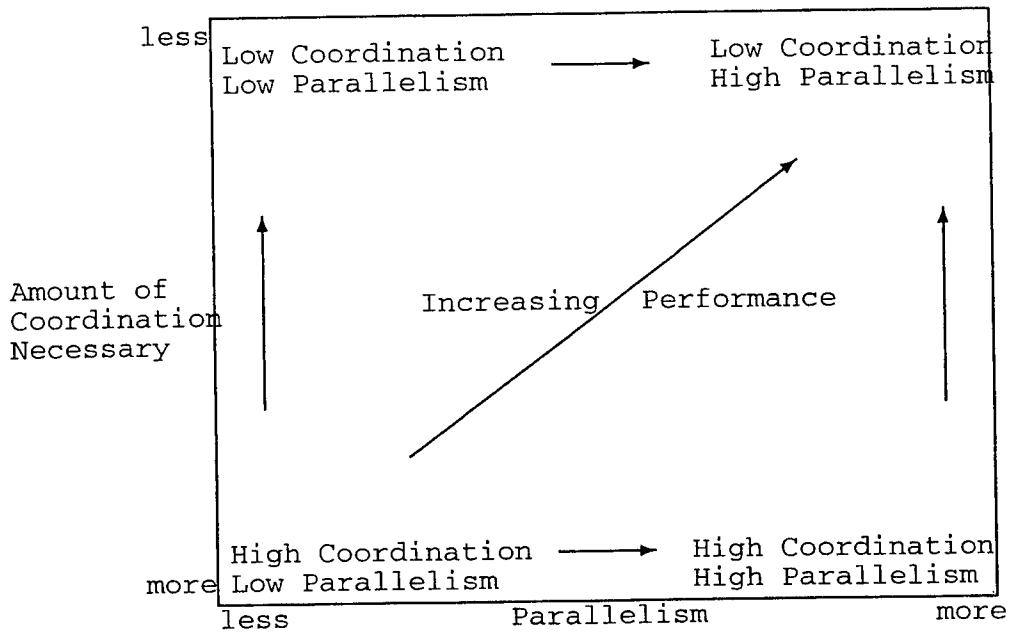


Figure 4.38: Variance in performance among distributions.

for an increase in performance this was a desirable side effect.

DARES achieved much of its performance by pruning its search space through the distribution of knowledge. By using coordination and communication heuristics to maintain beneficial distributions a set of DARES agents was able to reason much better than a single agent. Our experiments with SHYRLI have shown us that hyper-resolution has already pruned the search space to such a degree that distributing the axioms incurs almost no further benefit. Some benefit is gained in that the agents have fewer axioms with which to attempt resolution. SHYRLI, however, significantly outperforms DARES.

Our experiments have shown us that SHYRLI agents perform functional roles in relation to their nuclei. Each agents role is defined by their nuclei. The behavior that the communication and coordination strategies engenders is one that when an agent determines it is no longer making progress it attempts to import anything that will allow it to continue processing. The type of knowledge that will allow an agent to continue processing and the type of deductions that an agent can make are defined by an agents nuclei.

4.4.3 Differences between Single Agent and Multiple Agent Search

In order to better understand the difference between the way the multiple agents and the single agent perform search, we give the following examples. Figure 4.39 shows the deduction tree for one distribution of the clauses shown in Figure 4.22. This is the same set as in the

example given in Chapter 3. The deduction tree for the distribution given in Chapter 3 can be seen in Figure 4.12. As before, circles represent given axioms and the negated theorem, boxes represent the derived clauses. The electrons that were derived earliest are higher in the tree than electrons that were derived later.

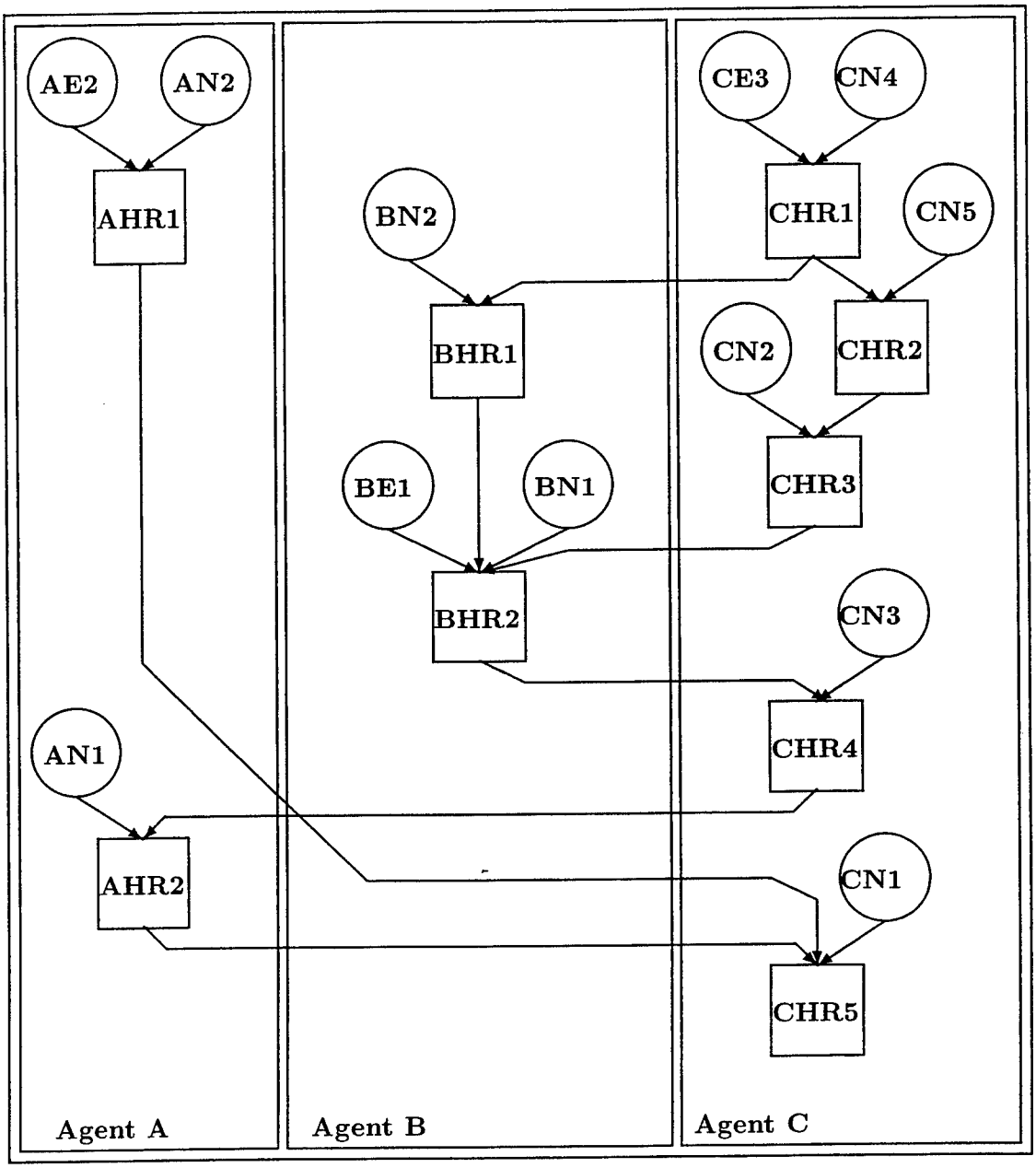


Figure 4.39: Deduction tree for the three agent example.

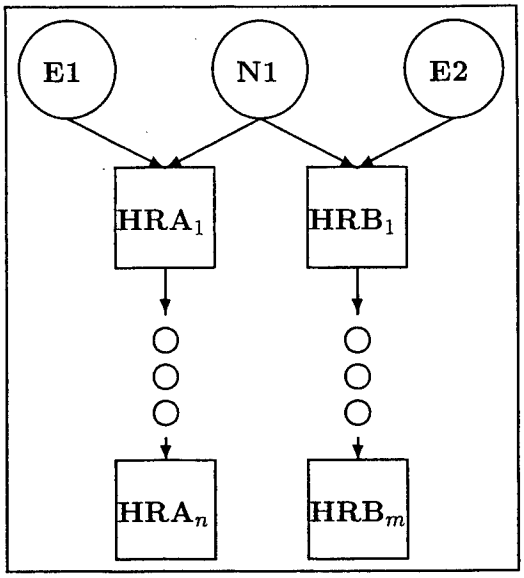


Figure 4.40: Single agent deduction tree example.

When comparing these two trees notice how **BHR1** is derived earlier in time in Figure 4.39 than in Figure 4.12. This is due to the cost of communication. In the example in Figure 4.12 we modeled a higher cost of commmunication than in Figure 4.39. In the example in Chapter 3, Agent C replied to Agent B's first request for assistance with **CHR1** and **CHR3**. In the example given in Figure 4.39 Agent B's first request resulted in a reply from Agent C with the electron **CHR1**. Agent B then made a second request in which it obtained **CHR3**. This faster communication resulted in somewhat more communication and more tightly coupled the agents.

As can be seen from Figure 4.12 and Figure 4.39 in some cases the deductions in the multiple agent case do not neccesarily occur in the same order as in the single agent case. In comparing the multiple agent case to the single agent case at an instance in time, some lines of deduction in the tree are longer and some are shorter than the single agent case. This can be due to the cost of communication as above or the distribution.

As an example of this phenomena, observe the case in Figure 4.40. Notice that there are two lines of reasoning that the single agent pursues. Each level of deduction in both lines are produced at the same time. The search is at the same level for both lines of reasoning at a given instance of time.

Now examine the multiple agent case in Figure 4.41. Notice that agent B will follow one line of reasoning until it determines outside assistance is necessary. This can occur when either the line of reasoning can proceed no further (because other agents hold neccessary knowledge or the line is a dead end) or the forward progress heuristic is not satisfied. In

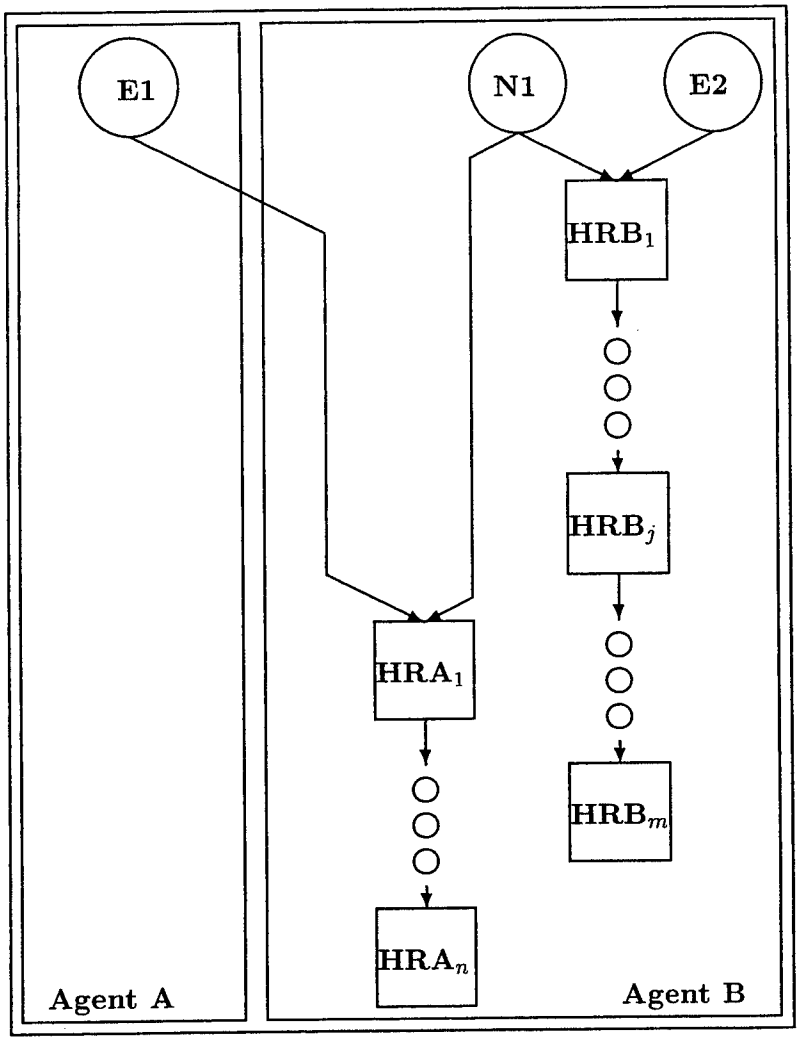


Figure 4.41: Multiple agent deduction tree example.

Figure 4.41 when Agent B's forward progress heuristic is not satisifed when hyper-resolvent $\mathbf{HRB}_j$ is generated. At this point Agent B asks Agent A for help and receives as an answer electron **E1**. Now Agent B can begin pursuing the second line of reasoning. The deductions in line A and line B are now offset in time by j.

It can be seen in this example that the multiple agent case will not necessary find the same proof as the single agent case. If both lines of reasoning lead to a contradiction, the single agent case will find the contradiction that is closest to the top of the tree, for that is the nature of breadth first search. If the closest contradiction exists in reasoning line A, then the multiple agent case in this example might find the contradiction in line B, as the agent has pursued reasoning line B further than the single agent case.

4.5 Results Analysis

The motivation behind this research was the inefficiency of the binary-resolution used in DARES [56]. With SHYRLI we have extended the architecture of DARES to incorporate a stronger inference rule, hyper-resolution. In doing so it became necessary to modify or replace many of the heuristics that guided the theorem-proving system. These included the heuristic for assessing local performance and the heuristics for formulating questions and answers.

The use of hyper-resolution in SHYRLI has allowed us to apply the architecture proposed in DARES to problems of more practicality. Binary-resolution was so inefficient that it could only be applied to small toy problems. At the moment SHYRLI is being used in ongoing research into generic multi-agent coordination schemes [37].

We have used specific aspects of the hyper-resolution inference rule to develop mechanisms to identify the information that does not have to be communicated in problem solving. This allowed us to develop the notion of private knowledge that an agent does not have to share. Knowledge that can be kept private can be identified in a domain independent manner. These privacy mechanisms guarantee that keeping specific information private will not affect the problem solving activity of the group of agents.

SHYRLI's behavior is substantially different from DARES. Although SHYRLI outperforms DARES, SHYRLI's single agent case usually outperforms multiple agent cases for small problems. As the size of the problem increases SHYRLI's multiple agents case begins to outperform the single agent case. This was not the case in DARES. The DARES reasoning system took advantage of a large unpruned search space. DARES was able to prune its search space using the distributions of knowledge. By incorporating heuristics DARES was successfully able to continue to maintain a fairly good partition of knowledge when communicating between agents. These distributed sets of knowledge produced fewer resolvents than the single agent case increasing the efficiency of the multiple agent case.

SHYRLI's use of hyper-resolution means the advantage of gaining a pruned search space through a distribution of knowledge is lost. Hyper-resolution has already significantly pruned the search space. SHYRLI's sensitivity to different distributions is a result of a distributed set of agents trying to traverse a search tree with a very limited number of paths to the goal (or contradiction).

DARES concluded that although some distributions affected DARES performance, these cases still outperformed the single agent case significantly. Therefore it was not important how the knowledge was distributed. The results we have obtained with SHYRLI seem to contradict this. Different distributions can significantly affect the performance of the reasoning system.

When we originally began the experiments described in section 4.4 we were sure that SHYRLI agents would request assistance more often than DARES. For the general behavior

experiment described in DARES, SHYRLI coordinated about the same amount. This means that we are achieving a speed up in performance without an increase in communication cost. As this small problem took DARES to its limits we could not compare the results of our larger problems with DARES.

SHYRLI agents have a functional role. Because SHYRLI agents keep their nuclei private, the agents with specific nuclei assume specific roles in the proof of a theorem. Using this inherent functionality we have developed a formalism in the predicate calculus that allows agents with different specialties to coordinate their activities. This predicate calculus formalism allows a complex problem to be decomposed and then distributed over a set of agents.

Chapter 5

Distributed Constraint Based Planning

5.1 Introduction

Interaction. This is the key ingredient, the very essence of what distinguishes *Distributed* Artificial Intelligence (DAI) from Artificial Intelligence (AI). Interaction is what drives the formation of architecture, plan and environment representation, and search control - the three major factors involved in developing a model for semi-autonomous agent behavior and an associated planning system. In the following paragraphs, we will first describe the requirements of these three components of what we term a *cooperative* planning system. Then we will outline our approach for meeting these requirements and we conclude by denoting the major contributions and features of our research.

Multiagent interaction can take on various and diverse forms. In some instances, agents work as equals cooperating to develop solutions[13]. In other cases, agents behave in a more managerial way with one or more agents directing the actions of other subordinate agents[6, 16, 31]. Sometimes a centralized problem is distributed with the hope of improving performance, in which case agent interaction is freely designated by the problem designer. Often the motivation in a case such as this is to improve the speed of problem solving. In other cases, time is not a factor. Instead it may be desirable to model an inherently distributed problem, in which case agent interaction must conform to an existing structure. As a result, a model for semi-autonomous agent behavior must have a flexible architecture so that various forms of agent interaction may readily be represented[75, 22, 28, 21].

Plan and environment representation must be sufficiently rich to capture all aspects of interaction in a complicated multiagent world. Of critical importance are aspects of time and temporal relations concerning agent interactions. Some interactions may require simultaneous actions of multiple agents. Other actions may be decoupled so as to have no

temporal relation at all, and thus multiple plan executions representing various orderings of these decoupled actions should be permitted.

In giving agents some degree of autonomy, the system must allow for flexible and diffuse control of search strategies. An agent should be able to take responsibility for task decomposition, either through its own actions or by directing the actions of others and collecting the results. It should also be possible for an agent to ask other agents to assume the responsibility of task decomposition. Decisions for when and how agent interactions occur should take into account agent work load, local availability of required resources to complete the plan, and local availability of task decomposition knowledge. Effective coordination of semi-autonomous agents should result in a balance of planning responsibilities and a coherent reorganization of the problem while minimizing agent interference.

With these factors in mind, we have developed a cooperative planner called DCONSA - Distributed Constraint-based planner for Semi-autonomous Agents. We call DCONSA a cooperative planner because it builds upon the expressive power of representational frameworks developed for multiagent planning combined with the coordination strategies found in distributed planning. In section 5.2 we give an overview of the planning literature covering many of the basic concepts of planning, Distributed AI, and constraint satisfaction problems. We also discuss how DCONSA relates to and expands this work.

DCONSA uses a plan and environment representation that embodies the intent of the GEM model [45]. As Lansky has shown through several examples, the GEM model is sufficiently rich to capture the many aspects of interaction, including critical temporal aspects. GEM explicitly recognizes the need to exploit constraint localization, and this concept plays a major role in the search strategy of DCONSA. Understanding the GEM model is fundamental to understanding DCONSA, therefore, we cover the basics of this model in section 5.3.

In section 5.4, we discuss how one can create a distributed architecture for distributed planning by taking a different view of the GEM model. We illustrate the flexibility of this distributed architecture through the introduction of the Multi-table Blocks World.

The major portion of our work has involved the development of a coherent distributed planning process that incorporates multiple parallel searches and agent autonomy. In section 5.5, we explain in detail how we start with a localized search process and add mechanisms to distribute that search. We then discuss how parts of this search can be conducted in parallel by planning for conjunctive goals. This parallel search is then followed by a serial search that combines these plans into one overall coherent plan. In the last part of this section, we discuss how agent autonomy is incorporated into this distributed, parallel search. This is accomplished by allowing agents to select from among three interaction modes. This selection is made mutually between two agents through a simple negotiation. The criterion for accepting a proposed interaction mode is based upon a heuristic measure of the agents' respective work loads. By basing this decision on relative work load, agent's

can interact in a distributed, dynamic fashion that balances the planning work load and reduces agent interference.

Section 5.9 presents the experimental work we have conducted using DCONSA. Through empirical results, we show that when agents use the heuristic decision criterion described in section 5.5, the planning work load is more balanced, thus increasing the amount of planning work performed in parallel. The result is an improvement in the time to construct a plan.

5.2 Planning, The Constraint Satisfaction Problem and Why DCONSA?

Perhaps the best method for describing the motivation for the development of DCONSA is to look at the progress of planning within the AI field. In this section, a partial historical overview of the planning literature is presented[38]. Included in this description are the origins of planning, various modifications resulting from a maturing field, and recent work which is directly related to our work. The section concludes by illustrating how DCONSA follows as a logical step in the development of the field.

5.2.1 History of Planning

5.2.1.1 Defining a New Field

Since the early 1970's, researchers have been intrigued with the notion of a computer program which could determine the steps required to achieve a desired outcome. This research effort became known as *planning* and remains an active field of AI to this day.

One of the earliest planners was STRIPS[25]. This planner set the model for many future planning systems and defined many of the terms and classic problems of the field. STRIPS is a linear state-based planning system which determines plans in the following manner. The planner is given a description of an initial *world state* and a *goal state* using first order predicate logic statements. The planner is also given a set of *operators* which can be used to modify the state of the world. The operators modify the world state by adding and deleting first order predicate logic statements used to describe world state. The statements which are effected by an operator are enumerated in the *add-list* and *delete-list* of that operator. Before an operator may be applied, the planner must decide if it is possible for the operator to be used. For instance, in order to walk into a room, the door to the room must be open. This is accomplished by ensuring that each statement in the operator's *preconditions* is true before the operator is applied.

The planning strategy of STRIPS involves *forward chaining*, that is, by examining the initial world state and the goal state, STRIPS tries to find a sequence operators that will

transform the state of the world from the initial state to the goal state [1]. In selecting an operator, STRIPS measures the difference between the current world state and the goal state and chooses an operator that reduces this difference. This strategy, known as *means end analysis*, was first presented in [24]. If the required operator cannot be applied because certain preconditions are not true, STRIPS establishes achievement of the unsatisfied preconditions as temporary intermediate goals.

If more than one goal is described in the goal state, then *conjunctive goals* exist and STRIPS will arbitrarily order the goals and attempt to achieve them serially. For this reason, STRIPS is termed a *linear* planner. The assumption made, termed the *linearity assumption*, is that the achievement of conjunctive goals is a decoupled problem. A problem arises, however, if the solutions of conjunctive goals are not decoupled. The arbitrary ordering of goals can not only effect the efficiency of the planning system, it can even determine whether or not a solution is found.

A partial solution to this problem was presented by ABSTRIPS[72] through the use of *hierarchical planning*. Levels of criticality were introduced into the STRIPS planning model. A criticality was assigned to each goal and to the preconditions of each operator. Thus, a plan could be constructed by working from the most abstract or general level to the most complete or detailed. However, since no backtracking between levels was allowed, there were still problems for which no plan could be found.

5.2.1.2 Planning with Partial Orderings

NOAH was the first planner to introduce the concept of planning with partial orderings [73]. Rather than viewing a plan as a linear series of operators, NOAH introduced the *procedural net*. A procedural net represents a plan as a network (graph) of planning states related by a partial ordering. The search strategy involves developing a plan through successive changes to the network of states. This alleviated the problems associated with placing an arbitrary order on the solutions of conjunctive goals. NOAH was a hierarchical planner as well, although it did not use criticality assignments. Instead, NOAH allowed nodes in the procedural net to represent different levels of detail. Nodes are initially abstract and represent a skeletal plan. As planning progresses, abstract nodes in the network are expanded to represent more detailed actions.

Some later planners were essentially modifications to NOAH which made its search more efficient. NONLIN [82] used a heuristic to rank the possible expansions of a node. The expansion which locally looked the most promising was chosen, and the other expansions were saved in case of required backtracking. NONLIN also introduced a goal-state table which explicitly represented the dependency between actions and the goals which they were attempting to achieve. SIPE [87] also used this notion and added the use of limited

[1] This is opposed to *backward chaining* in which a planner starts with the goal state, finds an operator that achieves the goal state, and then attempts to satisfy the preconditions necessary to apply that operator.

resources as a means of pruning search. DEVISER[86] is another planner which used the NOAH model of planning. DEVISER introduced the use of deadlines and time windows for goals and actions to improve the search process.

5.2.1.3 Constraint-based Planning

MOLGEN[79, 80] is another hierarchical planner which introduced new ideas for managing the solution of conjunctive goals. The interactions between subproblems in MOLGEN are represented explicitly as *constraints* which must be satisfied in order for a plan to be successful. Variables included in these constraints could be instantiated immediately or instantiation could be delayed until sufficient information in the form of additional refining constraints was gathered. The concept of delaying variable instantiation until absolutely necessary is called the *least commitment strategy* and is used to avoid early uninformed decisions which could possibly lead to backtracking.

MOLGEN also introduced a new form of hierarchical planning. Not only did MOLGEN plan by using various abstractions of operators, it also planned *how* it was going to plan. By incorporating a level of planning which selected from various planning strategies, MOLGEN introduced the idea of *meta-planning.*

5.2.2 Planning and Distributed AI

After MOLGEN and the NOAH family of planners, work in the planning field branched into many different research directions. Emphasis was placed on particular aspects of planning depending upon individual research interests. These varying interests include plan recognition in natural language processing, reactive planning, real time planning, execution monitoring, etc. This section is devoted to recent work in planning in *Distributed* AI (DAI), for it is this work that is most closely related to DCONSA.

Planning in DAI takes on several different meanings including systems with one or more of the following characteristics:

- Distributed world state information
- Distributed planning knowledge
- Distributed system control
- Plan execution by multiple agents
- Plan construction by multiple agents

Each of the systems described in this section is considered a part of DAI since each has at least one of the above characteristics. So, although the emphasis of the work presented

varies greatly, each makes a significant contribution to aspects of planning in DAI and is therefore related to DCONSA.

5.2.2.1 Distributing a Traditional Planner

One of the first research efforts in planning in DAI involved distributing a well known planner, NOAH. Distributed NOAH[16] used multiple agents to construct a plan, uniform planning knowledge at each agent, and investigated the passing of skeletal plans between agents. Control of plan construction is directed by agents that parcel out node expansion to other agents. Once node expansion is completed, the subordinate agents return the resulting expansion to the initiating node.

5.2.2.2 Cooperation in Planning

When multiple agents are involved in constructing a plan, an immediate concern is the organization of these agents. What will be the form of the architecture that defines their interrelationships? Will they act as equals or will a more managerial architecture be enforced with one or more agents directing the actions of other subordinate agents? What will be the cooperation strategy?

These questions were the focus of experiments in various cooperation strategies for air traffic control[6]. This research pointed out that cooperation strategies fall into two categories, organizational policies and information-distribution policies. Organizational policies are concerned with how a random network of agents takes on a fixed architecture at least for one task. Information-distribution policies determine strategies of inter-agent communication. These policies consider whether information is broadcast to all agents or directed to specific agents, whether information should have to be solicited or automatically offered, etc.

Other work in cooperation has concentrated on the efficient use of distributed resources for achieving multiple goals [13, 44, 14]. A protocol for information exchange when agents work as equals controlling distributed system resources was developed. A major focus of the work is discovering what minimum information is required for transmission among the agents. Agents use local information relating alternative plans with local resource usage to determine sets of local alternatives which are incompatible. This information is sent to other agents whose alternatives are tied to the local decision process. This information is then abstracted to the level of goals to determine incompatible goals. In this manner, agents exchange abstracted information regarding local resource conflicts to determine sets of goals whose achievement is mutually exclusive.

5.2.2.3 Modeling Multiagent Actions

As planning efforts in DAI increased, it became apparent that a richer plan and environment representation was needed to express the various types of interaction that occur in multiagent domains. This led to new models of representation that were explicitly developed to capture the interaction of multiple agents.

Process models[32] were introduced as a means of modeling the observable behavior of agents. Process models are essentially finite state graphs with state transition functions that may be dependent upon the state of other agents or the environment. As such, they can be used to model interaction between agents or an agent and its environment. They can be used to represent sequential as well as concurrent actions. Moreover, process models can be used to detect interference between the actions of multiple agents or an agent and its environment. This is essential in developing multiagent plans.

Another approach involves concentrating on the events that occur in a world model and their related constraints rather than focusing upon the transition of state. This method, introduced by the GEM[45] model, easily represents the complicated causal and temporal relations among the actions of multiple agents. Although process models emphasize agent actions over world state, they utilize a state-based framework. Concepts such as mutual exclusion, required simultaneity and priority requirements are commonplace in multiagent domains but are awkward to describe in a purely state-based approach. These types of constraints, however, are easily defined in an event-based model using first order predicate logic constraints. Another significant contribution of the GEM model is the localization of constraints to regions of activity. This bounds the constraints that must be checked at any point and guides the search process in planning.

5.2.2.4 Cooperative Distributed Problem Solving

Although cooperative distributed problem solving (CDPS)[19] is not strictly the same as planning for multiagent domains, it involves some of the same concepts and thus the two are directly related. The two major works in CDPS are the Contract Net Protocol[75] and PGP (Partial Global Plans)[22]. The main difference between this work and planning for multiagent domains is that in CDPS, the main objective is not to generate a complete plan. Rather, the concept of forming plans is used to guide problem solving.

In the Contract Net Protocol, an agent decomposes a problem into its subparts and posts notices of work it needs accomplished to solve these subparts. Other agents in the system place bids for the jobs depending upon their current capabilities and workload. After a time out period, the originator of the job (the contractor) awards the contract to the agent (bidder) with the best bid. A basic assumption in this work is that the subparts (subgoals in the planning context) are relatively independent and thus there is no concern regarding interaction among the subparts. The importance of this work is its ability to dynamically

change the architecture defining agent interaction. Agents will take on the contractor or the bidder role at various points in problem solving as the needs of the system change.

PGP has been developed for CDPS in a domain that involves analysis of distributed sensor data for vehicle track monitoring. The problem solving agents exchange local partial plans describing their current problem solving strategy. These are in turn used by the agents to form partial global plans which describe, at an abstract level, the planning strategy of several neighboring problem solvers. The partial global plans can then be used to focus and guide cooperative behavior among these agents. As indicated by the name, PGP does not create a complete plan with all interactions and details resolved. The intent is to use an abstract and partial view of agent problem solving state to steer future problem solving down a productive path.

5.2.3 The Constraint Satisfaction Problem

As noted in our discussions of MOLGEN and GEM, planning can be viewed as a constraint satisfaction problem. A distributed version of this approach is used in DCONSA, so we will now define the constraint satisfaction problem, and discuss related research in distributed constraint satisfaction.

The constraint satisfaction problem is defined as follows. Given a set $V = \{V_1, \ldots, V_n\}$ of variables, a set $D = \{D_1, \ldots, D_n\}$ of domains, where each domain D_i is the set of possible values for the variable V_i, and a set $C = \{C_1, \ldots, C_m\}$ of constraints which define compatible instantiations of values for the variables in V, find a value for each variable in V such that the set of values found does not violate any constraint in C.

Yokoo et al. [93] present a distributed version of the constraint satisfaction problem where there is a one to one mapping of variables to agents. Each agent is responsible for assigning a value to its variable. The set of variables is fixed and only binary constraints are allowed. Constraint checking among variables belonging to distinct agents is accomplished locally and assignments which turn out to be incompatible are collected in a data structure. This data structure is passed to other relevant agents thus allowing nonlocal constraint information to be incrementally formed by the problem solving agents. An algorithm is presented utilizing this information passing which the authors say can readily be extrapolated to a multi-variable per agent model.

A distributed approach which does not restrict the constraint types or agent-variable mapping is presented in Sycara et al. [81]. In this work, each agent is assigned a set of variables for which it must find values consistent with a set of constraints. Agents make value assignments asynchronously with limited global information. The domain explored in this work was job shop scheduling. In this domain, agents are assigned orders to schedule (the variables). Orders consist of activities which use system resources (the values). Some

resources can only be assigned to activities by a single agent while other resources can be assigned by multiple agents. These are termed, local and shared resources respectively. Many heuristic approaches are employed to improve local assignments and lessen the need for backtracking. These heuristics include the exchange of demand vectors for shared resources and various value and variable ordering schemes.

One assumption made by these distributed approaches is that the set of variables and their possible values are fixed. While this is a valid assumption in many AI problems, it is not always the case in planning. Planning often involves the synthesis of new variables and new domains for these variables in the form of subgoal creation.

5.2.4 Introducing DCONSA

The development of DCONSA draws from the best ideas of these past research efforts and extends them in a new general framework for cooperative planning. DCONSA is an acronym for Distributed Constraint-based Planning for Semi-autonomous Agents. In this section, each of these phrases will be explained as well as how DCONSA incorporates these past research efforts.

Distributed - DCONSA has a general framework which allows the modeling of distributed world state information, distributed plan decomposition knowledge, and distributed system control or any combination thereof. With such flexibility, DCONSA is able to model agents which cooperate as equals[13, 44, 14], in a more managerial fashion [16], and experiments on a range of cooperation strategies are possible as well[6].

Constraint-based Planning - DCONSA uses a constraint-based representation [79, 80] with an emphasis on events and agent behavior [32, 45]. The planning process involves changes to an evolving graph representing actions and their interrelationships [73, 46].

Semi-autonomous Agents - DCONSA allows the agents developing the plan to have some degree of autonomy over their own actions and the role they play. Therefore, the architecture of the cooperation among agents is not fixed, but is dynamic and responsive to the needs of the system [75].

To summarize, DCONSA is a cooperative planner which permits investigation into each of the characteristics of DAI planning as listed in Section 5.2.2:

- Distributed world state information
- Distributed planning knowledge
- Distributed system control
- Plan execution by multiple agents
- Plan construction by multiple agents

5.3 The GEM Model

The GEM model [45] (Group Element Model) has been selected as the basis for our environment and plan representation. The GEM's emphasis on constraint localization - the association of constraints with explicitly defined regions of activity - serves three important purposes for the DCONSA system. Constraint localization directs the search process, bounds the set of constraints that must be checked at any point, and provides insight concerning the rules for distribution of environment information and planning knowledge. Since the GEM model plays such an important role in the DCONSA system, this section presents a thorough discussion of its concepts.

5.3.1 Basic Concepts

GEM is an event-based model for specifying the behavior of agents in multiagent domains. Events are used to represent occurrences in the world and the actions taken by agents. Each event is unique and should be considered atomic.[2] Events may be parameterized and they may also be grouped into event types. For instance, in the world of restaurants, there exists an event type of the form $Order(f : Food, w : Waitress)$ where a specific event may be represented by $order(steak, shirley)$. Throughout this discussion, types will be capitalized and specific instances will be represented with lower case letters. Unless a distinction is made, the word "event" will refer to both event types and specific event instances.

There are three event relations which can hold between any two events. These are: temporal order $\Rightarrow$, simultaneity $\rightleftharpoons$ and causality $\rightsquigarrow$. The temporal ordering is a partially ordered, irreflexive, antisymmetric, transitive relation between events. The simultaneity relation is a partially ordered, reflexive, symmetric, and transitive relation between events. The causality relation is a partially ordered, irreflexive, antisymmetric, and nontransitive relation between events. The causality relation is nontransitive because it models direct causality between two events. As such, causality necessarily implies a temporal ordering $(A \rightsquigarrow B \supset A \Rightarrow B)$. However, if one event is causally related to a second event, the occurrence of the first event does not imply that the second event must eventually occur. Lansky has defined the $\rightsquigarrow$ relation to have the sense of "enabling". If two events, A and B, have a causal relationship, $A \rightsquigarrow B$, then every event B must be enabled by the occurrence of an event A. However, if every event A must eventually be followed by an event B, then this must be explicitly defined in the domain description. This can be accomplished with a first order predicate calculus statement of the form: $\forall A \exists B : A \rightsquigarrow B$. This sense of the causal relation is adopted in the DCONSA system.

Events are grouped into regions of activity called *elements* and *groups*. Elements identify regions of sequential activity - i.e. events which are members of the same element must be

[2] If reasoning about temporal intervals is to be modeled, a single event may be comprised of a starting event and an ending event.

sequentially ordered. Every event must belong to a unique element. Groups, on the other hand, are used to identify regions of causal or required simultaneous activity. Groups consist of elements (and consequently their member events) and other groups. There are no limitations set upon the structural relationships among the regions that groups identify. These regions may be disjoint, strictly hierarchical, or they may overlap to varying degrees. As with events, elements and groups may likewise be collected into element and group types. The same naming conventions will apply.

Groups not only identify regions of causal activity, they also limit "causal flow". The structure formed by their interrelationships imposes implicit causal restrictions between events. Causal relations are directed relations. Causal relations may be directed or "flow" from within a group to events outside its group. However, causal relations may not flow from outside a group to the events within its borders. A simple analogy is the scoping of variables in the structure of programming code. [3]

Figure 5.1 illustrates some of the concepts presented thus far. Elements are enclosed by curved boundaries and groups by straight boundaries. The areas controlled by two assembly line robots are represented by groups *g1* and *g2*. The "realm" of robot1 includes a robotic arm, *arm1*, and two assembly tools, *tool1* and *tool2*. The realm of robot2 includes a robotic arm, *arm2*, and one assembly tool, *tool3*. Since robotic arms perform actions in a sequential order, they are modeled as elements. The areas controlled by the assembly line robots are modeled as disjoint groups because events involving their respective tools can only be causally related to events (actions) of their respective robotic arms. Group *g3* models the coordinated activities of *arm1* and *arm2* such as the simultaneous actions required in passing a product between the arms. The *alarm* is modeled as an element as well, since its "ringing" events must be sequentially ordered. The *alarm* element is not placed in a group with any of the other elements because it cannot cause any of the events associated with these elements to occur. Note however, that there is no group that encompasses the *alarm* element. Thus, there is no restriction on causal flow directed from the other elements to the *alarm*. This allows events (e.g. accidents) involving the robotic arms to cause the alarm to ring.

5.3.2 Execution Modeling

Using these basic concepts, a structure is required which models the possible execution of specific events in a defined environment. This structure is called a world plan. A world plan, W, is a formal structure containing a set of event instances, a set of element instances, a set of group instances, the interrelationships among events, and the transitive subset relations

[3] To simplify this notion, a causal relation may exist between two events only if they belong to a common group in DCONSA.

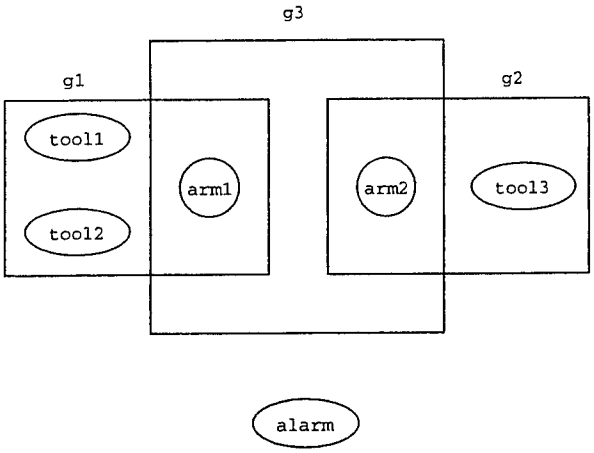


Figure 5.1: Robotic Assembly Line World

among events, elements, and groups.

$$W =< E, EL, G, \rightsquigarrow, \Rightarrow, \rightleftharpoons, \in >$$

- E = A set of event instances.
- EL = A set of element instances.
- G = A set of group instances.
- $\rightsquigarrow$: $(E \times E)$ The causal relation between the events. It is a partial, irreflexive, antisymmetric, nontransitive relation.
- $\Rightarrow$: $(E \times E)$ The temporal ordering among the events. It is a partial, irreflexive, antisymmetric, transitive ordering.
- $\rightleftharpoons$: $(E \times E)$ The simultaneity relation between the events. It is a partial, reflexive, symmetric, transitive relation.
- $\in$: $(E \times \{EL, G\})$ The membership relation between the events and the elements or the events and the groups.

A world plan is the "upper tier" of a two level model. At the lower level, one can view a world plan as all the possible executions of that plan. When no explicit temporal ordering exists between two events in a world plan, the lower level interpretation of that world plan will include a plan for each possible ordering of those two events. The world plan is thus a compact representation of many plans.

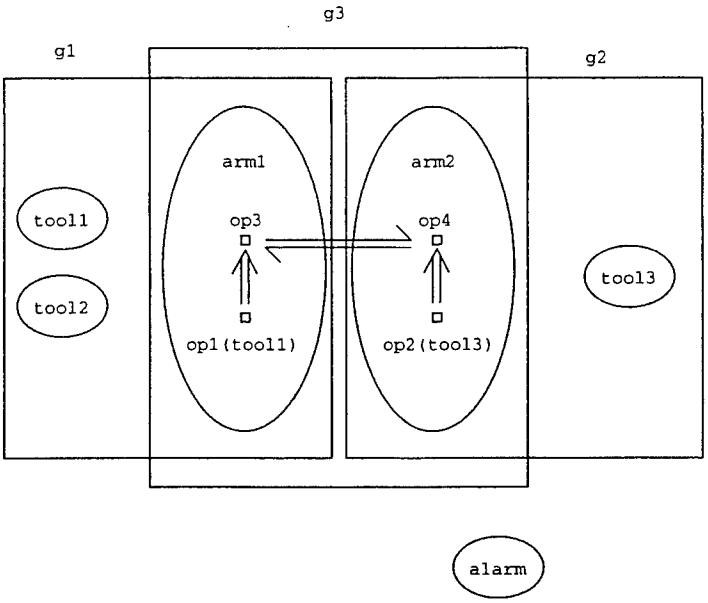


Figure 5.2: Example World Plan

Returning to the assembly line robot example, consider Figure 5.2 which is a graphical representation of a possible world plan. As before, elements are represented with curved boundaries and groups are represented with straight boundaries. Events are represented graphically as small squares. This world plan models the execution of four events: $op1(tool1)$, $op2(tool3)$, $op3$, $op4$. The events $op1(tool1)$ and $op2(tool3)$ can be interpreted as assembly operations involving locally available tools. These local operations must be completed before two coordinated events, $op3$ and $op4$, occur.

Since there is no temporal relation between events $op1(tool1)$ and $op2(tool3)$, there are three possible executions of this world plan:

1) 1st $op1(tool1)$ 2nd $op2(tool3)$ 3rd $op3$, $op4$,
2) 1st $op2(tool3)$ 2nd $op1(tool1)$ 3rd $op3$, $op4$,
3) 1st $op1(tool1)$, $op2(tool3)$ 2nd $op3$, $op4$;

$op1(tool1)$ may occur before, after, or simultaneously with $op2(tool3)$.

Each of these possible executions is termed a *valid history sequence* (VHS). A *history*, α, of a world plan is a prefix of that world plan. A history is a set of events and their relations which have occurred up to a point in the execution of a world plan. If two events are temporally ordered, they must enter a VHS in different histories. On the other hand, if two events occur simultaneously, they must enter a VHS in the *same* history. To illustrate,

VHS 1 above has four histories: [4]

- α_0:{}
- α_i:{$op1(tool1)$}
- α_j:{$op1(tool1)$,$op2(tool3)$}
- α_k:{$op1(tool1)$,$op2(tool3)$,$op3$,$op4$}

A VHS is called *complete* if its first history is empty. There are three complete valid history sequences containing each event in the world plan depicted by Figure 5.2. Introducing α_l:{$op2(tool3)$}, these three VHSs can be described by:

S1: α_0 α_i α_j α_k
S2: α_0 α_l α_j α_k
S3: α_0 α_j α_k

Formally, a sequence of histories, $S : \alpha_0, \alpha_1, \ldots$ is a VHS if and only if [45]:

1. The sequence is monotonically increasing:

$$\alpha_0 \subset \alpha_1 \subset \alpha_2 \ldots$$

2. Temporally ordered events must enter the sequence in distinct histories:

$$(\forall \alpha_i \in S, i > 0)(\forall ej, ek \in \{\alpha_i - \alpha_{i-1}\})\neg[ej \Rightarrow ek]$$

3. Simultaneous events must enter the sequence in the same history:

$$(\forall ej, ek)[ej \rightleftharpoons ek \supset (\forall \alpha_i)[ej \in \alpha_i \supset ek \in \alpha_i]]$$

As stated earlier, elements and groups define regions of activity. The constraints on activities within these regions are described by user defined constraints. Two types of constraints are handled: first-order temporal logic formulas, and constraints expressed as regular expression patterns.

The regular expression constraints are used to express patterns of event occurrences in a domain. Constraints of this type can only be applied to a set of events which are totally ordered. A qualifying formula is used to determine which events are relevant to the constraint. These events and their relations are then checked against the pattern described by the regular expression of the constraint. This regular expression may use the normal

[4]For brevity, only the events are listed here. Remember, however, that each history also includes the relations among these events.

operators of concatenation, Kleene closure, positive closure, and union, over a string of events and event relations.

The first-order-temporal logic formulas may use the standard connectives and quantifiers: $\wedge$, $\vee$, $\neg$, $\supset$, $\Leftrightarrow$, $\forall$, $\exists$, $\exists!$ (unique existence). Event, element, and group variables may also be used in these formulas with domains of event, element, and group instances respectively. Nontemporal formulas may be applied to a single history (e.g. $\alpha \models Q$) [5]. Predicates which may be evaluated in these nontemporal formulas include:

$occurred(e)$	Event e is in the history.
$e \in EL$	Event e is in the history and is a member of element EL.
$e \in G$	Event e is in the history and is a member of group G.
$e1 \Rightarrow e2$	$e1$ and $e2$ are in the history and $e1$ occurs before $e2$.
$e1 \rightleftharpoons e2$	$e1$ and $e2$ are in the history and occur simultaneously.
$e1 \rightsquigarrow e2$	$e1$ and $e2$ are in the history and $e1$ causes $e2$.

First-order temporal logic formulas may be applied to history sequences using the linear-time temporal operators $\Box$ henceforth, $\Diamond$ eventually, $\bigcirc$ next, $\bigcup$ until, $\triangle$ before, $\overleftarrow{\Box}$ until-now, and $\overleftarrow{\Diamond}$ back-to. Note that VHSs have the tail closure property: if $S : \alpha_0, \alpha_1, \ldots, \alpha_{i-1}, \alpha_i, \alpha_{i+1}, \ldots$ is a VHS, then $S[i] : \alpha_i, \alpha_{i+1}, \ldots$ is a VHS. Given a history sequence, $S : \alpha_0, \alpha_1, \ldots$, these temporal operators are defined as [45]:

- P is *henceforth* true for a sequence $S[i]$ if P is true of every tail sequence of $S[i]$.

$$S[i] \models \Box P \equiv (\forall j \geq i) S[j] \models P$$

- P is *eventually* true for a sequence $S[i]$ if P is true for some tail sequence of $S[i]$.

$$S[i] \models \Diamond P \equiv (\exists j \geq i) S[j] \models P$$

- P is true *next* for a sequence $S[i]$ if P is true of the first tail sequence of $S[i]$.

$$S[i] \models \bigcirc P \equiv S[i+1] \models P$$

- P is true *until* Q for a sequence $S[i]$ if 1) P is true of every tail sequence of $S[i]$, or, 2) there is some tail sequence of $S[i]$, $S[k]$, in which Q is true and P is true of every tail sequence $S[j]$ where $i \leq j < k$. Note, nothing is stated about the truth of P for $j \geq k$.

$$S[i] \models P \bigcup Q \equiv (\forall j \geq i) S[j] \models P \vee (\exists k > i) S[k] \models Q \wedge (\forall j, i \leq j < k) S[j] \models P$$

- P is true *before* for a sequence $S[i]$ if P is true of the previous tail sequence of $S[i]$.

$$S[i] \models \triangle P \equiv S[i-1] \models P$$

[5] This is read: *"alpha models Q"*.

- P is true *until now* for a sequence $S[i]$ if P is true of every previous tail sequence $S[i] : S[0], \ldots, S[i-2], S[i-1], S[i]$.

$$S[i] \models \overleftarrow{\Box}\, P \equiv (\forall j, 0 \leq j \leq i) S[j] \models P$$

- Q *back_to* P is true for a sequence $S[i]$ if 1) Q is true until now $S[i]$, or, 2) there exists a sequence $S[j]$, $j < i$, such that P is true of $S[j]$ and Q is true of every sequence $S[j+1], S[j+2], \ldots, S[i]$. Note, nothing is stated about the truth of Q for $S[0], \ldots, S[j]$.

$$S[i] \models P \overset{Q}{\leftarrow\equiv\Box}\, Q \vee (\exists j, 0 \leq j \leq i) S[j] \models P \wedge (\forall k, j \leq k \leq i) S[k] \models Q$$

- A nontemporal formula Q is true for a sequence S if it is true of the first history of the sequence S.

$$S \models Q \equiv \alpha_0 \models Q$$

Constraints in the form of first-order temporal logic constraints can be applied to a world plan W by applying them to each of its complete, valid history sequences. The world plan W satisfies a constraint P, if and only if P is satisfied by *all* of its valid history sequences:

$$W \models P \equiv (\forall \text{ VHS } S \text{ of } W) S \models P \ .$$

5.3.3 Environment Representation

A GEM domain description is the representational definition of a particular problem's elements, groups, and the constraints on events associated with these elements and groups. Each element definition includes descriptions of the events which may occur within that element. Group definitions include descriptions of the elements and groups which they encompass. Also included in each element and group definition are domain dependent constraints. These localized constraints only pertain to those events which are associated within the scope of the element or group being defined (the relevant region). At any particular point in the planning process, only those constraints relevant to the current region involved need to be checked.

5.3.3.1 Element Definition

To reiterate, elements define regions of sequential activity. Elements will most often be used to model objects which can perform actions or make events happen. [6]The definitions

[6]In GEM, physical, inanimate objects are defined as elements that can perform no events. In DCONSA, the existence of such objects are denoted by logic statements.

of such elements must include an enumeration of the events associated with these elements. Also included in these definitions will be additional constraints on these associated events.

Returning to the assembly line robot example, the robotic arms, arm1 and arm2 may be defined in the following manner. For this example, we assume that every *Op3* must be preceded by an *Op1* and every *Op4* must be preceded by an *Op2*.

```
RobotArm = ELEMENT TYPE
EVENT TYPES
   Op1(t:ToolA)
   Op2(t:ToolB)
   Op3
   Op4
CONSTRAINTS
```

$(\forall e3 : Op3)(\exists! e1 : Op1)[e1 \Rightarrow e3]$ $(\forall e4 : Op4)(\exists! e2 : Op2)[e2 \Rightarrow e4]$

```
END RobotArm

arm1 = RobotArm ELEMENT
arm2 = RobotArm ELEMENT
```

5.3.3.2 Group Definition

Group descriptions are used to define regions of causal or required simultaneous activity. Thus, they can be used to model regions of directed or cooperative behavior. In addition, groups can be used to restrict causal flow among events in an environment. Thus, groups can be used to conceptually separate areas of independent activity.

As with elements, group type definitions may be used to define a form that several identical group instances may take.

To illustrate, consider the realms of the robots of the assembly line example and their cooperative behavior. Here we assume that every *Op3* must occur at the same time as an *Op4*.

```
RobotRealm = GROUP TYPE (a:RobotArm)
```

```
END RobotRealm

g1 = RobotRealm GROUP (arm1)
g2 = RobotRealm GROUP (arm2)

g3 = GROUP (arm1,arm2)
CONSTRAINTS
```

$(\forall e3 : Op3)(\exists e4 : Op4)[e3 \rightleftharpoons e4] \wedge (\forall e4 : Op4)(\exists e3 : Op3)[e3 \rightleftharpoons e4]$

```
END g3
```

5.3.3.3 GEMPLAN

GEMPLAN is Lansky's planner that uses the GEM model. DCONSA's localized search method is based upon that of GEMPLAN. Therefore, we will present GEMPLAN's localized search method in the following paragraphs. For a more detailed discussion of GEMPLAN's localized search, the reader is strongly encouraged to see [47].

5.3.3.4 The Localized Search Method

At the region level, the search method of DCONSA is based upon that of GEMPLAN; that is, search is viewed as a localized constraint satisfaction problem. The search is "localized" because when search occurs in a region, only those constraints associated with that region are checked, and only events and event relations belonging to that region (and its subregions) are considered. Goals and task decomposition knowledge are represented as constraints to be satisfied by "fixes".[7] A fix is the method by which a constraint is satisfied and may include the addition of events and/or temporal relations among existing events in a plan.

The general form of a local search in a region can be viewed as a loop. With each pass through the loop, some region constraint which is not already satisfied by the region plan is selected. Fixes for this constraint are applied until one is found which alters the region plan in such a way that the constraint is satisfied. [8] This loop is repeated until all region

[7] These are called "resolvers" in DCONSA.

[8] GEMPLAN allows the user to specify a search method with a depth-first search as the default. DCONSA has been implemented with a depth-first search.

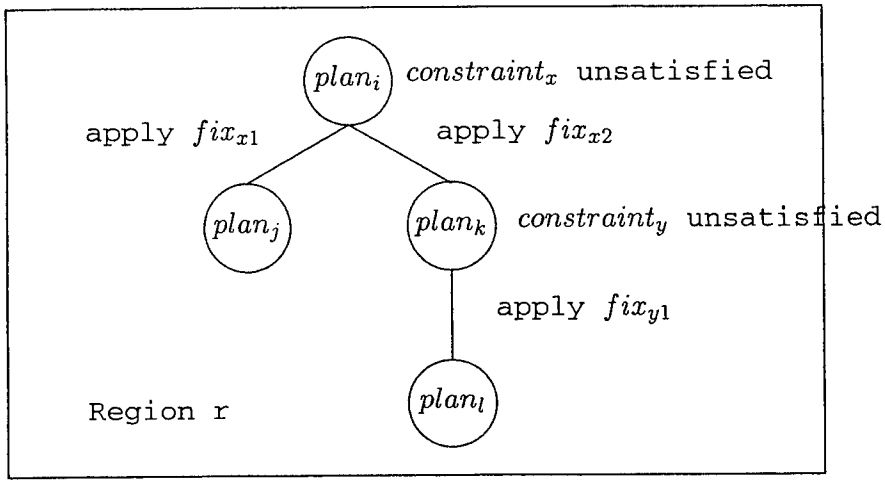


Figure 5.3: Example of a Local Search Space

constraints are satisfied. Thus, a region search space is a tree of nodes in which each node represents an attempt to verify the region constraints given the current region plan (see Figure 5.3). Each branch in the region search space represents the application of a fix to bring about modifications to the region plan. The branching factor of the region search space depends upon the number of possible fixes for each constraint.

The fixes used to satisfy constraints add events and event relations to the plan. Search must also extend to regions which could possibly be affected by plan modifications because the search process must ensure that the local region constraints of these affected regions are satisfied as well. For example, if the resolver includes adding an event to a subregion, that subregion must be searched. To illustrate, consider the graphical representation of the region *g1* in Figure 5.1. If during search in *g1*, a fix is applied which adds an event to *arm1*, the plan of *g1* is obviously modified. Thus, the constraints in *g1* must be checked against the new plan.

In addition, once search is completed in a region, the regions which encompass that region must be explored. This occurs because the plan of a region is a component of the plans in each "super"-region which encompasses it. Furthermore, information about a region's plan must be updated in its encompassing regions before search can proceed there. Returning to our example using Figure 5.1, the region plan of *arm1* is a component of the region plans of *g1*, and *g3*. Thus, any modification to the region plan of *arm1*, also induces changes in the plans of *g1* and *g3*. Such a modification must be reflected in the region plans of these super-regions and these new plans must be checked with their respective region

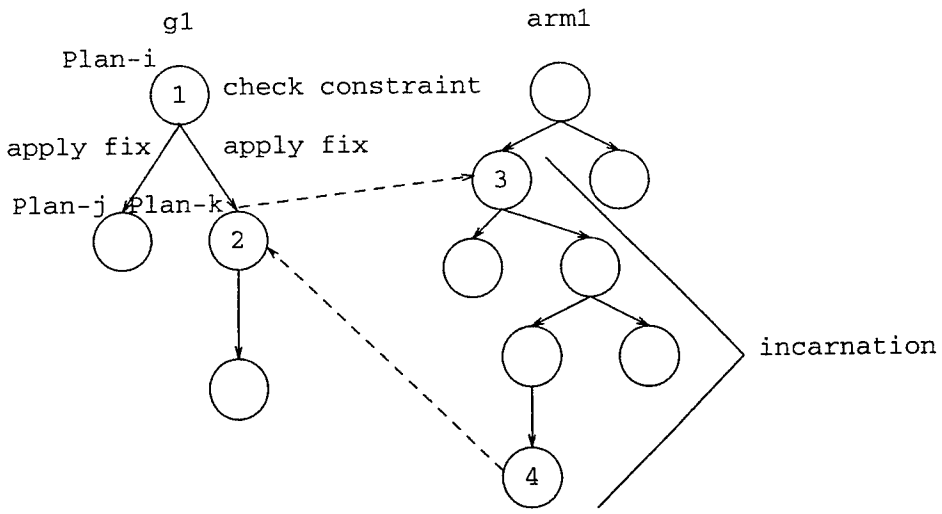


Figure 5.4: Incarnations in the GEMPLAN Search Space

constraints.

Thus, the search process for a plan, can be viewed as a series of region searches in region search trees. This process continues until the constraints in every region are satisfied or until all fixes have been tried.

During the search for a plan, the local plan of a given region may be visited multiple times. Each visit of a search to a region is called an *incarnation* of that search. Each incarnation will produce its own portion of the search space within that region. When backtracking occurs in a region, it is confined to the current incarnation.

To illustrate, consider Figure 5.4 showing an example of search spaces in group *g1* and Arm *arm1*. In this diagram, the application of the fix that transforms *g1's* plan from Plan-i to Plan-k, involves the addition of an event(s) to the plan of Arm *arm1*. This induces an incarnation of search in *arm1* beginning at node 3, the last node of the previous incarnation, and completing at node 4. Upon completion of this incarnation of search, search will continue in *g1* at node 2. If a failure occurred during the incarnation of search induced in *arm1*, the search process would backtrack as far as node 3, the head of the incarnation, and then the backtracking would stop in *arm1*. Backtracking would then proceed at node 2. Thus search flows through the search trees in the proper reverse order during backtracking.

This concludes our discussion of the basic points of the GEM model and its related

planner, GEMPLAN. In the following sections, we will begin our discussion of how DCONSA uses these concepts as a basis and expands upon them.

5.4 Creating a Distributed Planning Architecture

DCONSA is designed to develop plans in a distributed environment. This environment consists of semi-autonomous agents, each having local knowledge and control of local resources. These agents must be able to coordinate individual choices and actions so as to cooperate in achieving global goals. We achieve this coordination by organizing the domain knowledge in a formal structure which is distributed across a set of planning agents.

In this section, we describe how we can exploit the GEM model to create a distributed architecture from a formal organization of the planning problem. This formal organization is not dependent upon the number of existing planning agents. Thus, using this formal organization, we can vary the number of agents used in the distribution. Furthermore, the mapping of the organization to a fixed set of agents can be varied as well. This means we can use DCONSA to model different planning scenarios without changing the underlying problem representation. This flexibility is one of the contributions DCONSA makes to the area of Distributed Planning. In the DVMT [20], the mapping of problem representation to problem solving agents is tied to geographic considerations. MACE [28] leaves problem representation up to the user.

5.4.1 Looking at Regions in a Different Light

In GEM, regions are used to improve the centralized search for a plan by focusing the constraint satisfaction problem. Regions accomplish this by ensuring that constraints are checked against only those portions of the plan which are relevant to them. Without regions, all constraints would be checked against the entire plan. In DCONSA, regions conceptually serve a second purpose. Regions can be viewed as a means of organizing the constraint satisfaction problem into a distributed planning architecture.

When viewed in this way, domain knowledge, formulated as a constraint satisfaction problem, is organized using the concept of a region. In this organizational view, a region is a conceptual means of encapsulating a set of events with a set of constraints that relate those

events. An element, in effect, becomes a primitive unit of organization, since every event belongs to a unique element. The set of constraints which pertain only to the events of a single element are included as part of the element description. To reason about constraints which relate events in distinct elements, we need to incorporate a higher level of organization - a group. Remember, a group is a set of regions (elements and/or groups) and a set of constraints which relate events in its member regions. Thus, by using primitive and higher order units of organization, i.e. elements and groups, a planning problem can be structured as a set of regional constraint satisfaction problems.

Using this view of regions as organizational units, we can represent distributed domain knowledge in a formal way by distributing the given set of region definitions in a domain description to a collection of planning agents. To create a flexible planning architecture, we do not restrict this distribution to a "one region per agent model". [9] On the contrary, an agent may plan for any number of the regions included in a domain description. [10] Each group definition must include an enumeration of its subregions, but the definitions of these regions are not required to exist at the same agent as the group definition. Similarly, each region definition must enumerate the regions which encompass it, its "super"-regions, but again, these regions are not required to exist at the same agent as their common subregion.

5.4.2 The Multi-table Blocks World

In this section, we introduce the Multi-table Blocks World [11] to demonstrate the definition of a domain specific problem and its distribution among planning agents. The Multi-table Blocks World consists of multiple robotic arms working over multiple table surfaces. A robotic arm may have sole access to a table surface, or it may share access to a table surface with a number of other robotic arms.

The actions of an arm over a specific table surface must be fully temporally ordered. Therefore, we define this region of activity as an element. Furthermore, we define two types of events which can be executed by an arm over a table surface: $Pick(X)$ and $Put(X, Y)$.

[9] Lansky suggests that GEMPLAN, her GEM related planner, could be distributed[46]. However, from the context of the statement it is evident that the flavor of the suggested distribution is different from that of DCONSA. Lansky's suggested distribution involves distributing each regional search space to a separate processor. This distribution still involves an essentially centralized control of search with agents having direct access to the local plan structures of other agents (through data inheritance) and a global view of world state.

[10] Note however, that if one agent plans for all the given regions, a centralized version of the problem is modeled.

[11] An extension of the Blocks World domain presented in [46] with multiple table surfaces.

Figure 5.5 shows an informal definition for a TABLEARM element type.

TABLEARM *Element Type*
EVENT-TYPES:$Pick(X)$,$Put(X,Y)$
CONSTRAINTS:

1. All $Pick(X)$ events and $Put(X,Y)$ events over the table surface must be temporally ordered.
2. All movement of the same block b, $Pick(b)$ and $Put(b,Y)$, must occur before anything is stacked on b, $Put(X,b)$.

Figure 5.5: Table-Arm Element Type Definition

Now that we have a means of representing the actions of a robotic arm over a table surface, we can model the robotic arm itself as a composite of its actions over each table surface that is within its reach. To accomplish this, we define an ARM group type whose member regions are tablearm elements. The definition of this group type is found in Figure 5.6. The constraints in the arm definition are similar to the constraints of the tablearm, but they differ in the following manner. Whereas Constraint #2 of the tablearm definition ensures that movement of a particular block over a single table surface must occur before anything is stacked upon the block, Constraint #2 of the arm definition ensures the same condition over multiple table surfaces.

ARM *Group Type*
CONSTRAINTS:

1. The arm must alternate $Pick(X)$ events and $Put(X,Y)$ events.
2. All movement of the same block b, $Pick(b)$ and $Put(b,Y)$, must occur before anything is stacked on b, $Put(X,b)$.

Figure 5.6: Arm Group Type Definition

We also define a TableRealm group which describes the actions of the various TableArms over a Table. A TableRealm encompasses a table surface and the set of tablearms that can reach that table surface. The definition for a TableRealm group type is shown in Figure 5.7. We need this region definition for three reasons. First, we define goal conditions in terms of configurations of blocks over particular table surfaces. Thus, we define a constraint that ensures that these goal conditions are met. We also need to ensure that the mutual use of a block by different arms over the same table is coordinated so that nothing is stacked on the block until after all movement of the block is complete. The third reason we need this region, is to determine whether blocks are clear before they are picked up or put down. In the case where multiple arms have access to a table surface, the "clear"-ness of a block is effected by the actions of all the arms that can reach that block. Thus to check this

condition, we need a region that encompasses all the actions of arms over a table surface - the TableRealm.

TABLEREALM *Group Type*
CONSTRAINTS:

1. All goals of the form $On(X, Y)$ are achieved.
2. All movement of the same block b by the arms over the table, $Pick(b)$ and $Put(b, Y)$, must occur before anything is stacked on b, $Put(X, b)$.
3. For every $Pick(X)$ event, X is clear before the $Pick(X)$ event occurs.
4. For every $Put(X, Y)$ event, Y is clear before the $Put(X, Y)$ event occurs.

Figure 5.7: TableRealm Group Type Definition

Continuing our description, we now define a region that describes the interaction of the arm with the blocks that are on the table surfaces within its grasp. Again, in the case of a table surface that has multiple arms over it, determining whether a block is within an arm's grasp depends upon the actions of other arms. Therefore, this region, called a RobotRealm, will consist of an Arm, the composite TableArm elements of the Arm, each TableRealm to which the Arm has access, and each TableArm and Table of the TableRealms. The definition is given in Figure 5.8.

ROBOTREALM *Group Type*
CONSTRAINTS:

1. For every $Pick(X)$ event, there must exist a $Put(X, Y)$ event such that $Pick(X) \rightsquigarrow Put(X, Y)$.
2. For every $Put(X, Y)$ event, there must exist a $Pick(X)$ event such that $Pick(X) \rightsquigarrow Put(X, Y)$.
3. For every $Pick(X)$ event, before $Pick(X)$ occurs, X must be over a table within reach.
4. For every $Put(X, Y)$ event, before $Put(X, Y)$ occurs, Y must be over a table within reach.

Figure 5.8: Arm Group Type Definition

For the purpose of illustration, we use an example that involves five arms and nine table surfaces (see Figure 5.9). Table *t1* can only be reached by Arm *arm1*, *t2* by only *arm2*, etc. Table *t12* can be reached by *arm1* and *arm2*, *t23* by *arm2* and *arm3*, etc. Goals in this example involve stacking blocks on various table surfaces and other blocks. The group and element organization is presented graphically in Figure 5.10. Element regions are represented by circles, and group regions are represented by polygons. To help distinguish

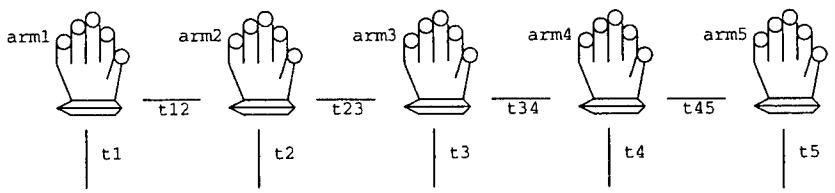


Figure 5.9: Multi-table Blocks World Example

various regions in this particular diagram, groups are drawn with solid, dashed, or dotted lines. These differing borders have no significance in this diagram.

5.4.3 Distributed Architectures for the Multi-table Blocks World

Many distributions for our example problem are possible. In the following paragraphs, we describe a few planning scenarios and show possible distributions to model them.

One can imagine modeling a situation where each robot plans its own actions and each table has a manager that coordinates the actions of the arms that work over its surface. This situation can be modeled with the distributed organization shown graphically in Figure 5.11. This distribution requires 14 agents, five for the RobotRealm groups and nine for the TableRealm groups. The dashed regions in the architecture represent regions whose relation with local regions is known (subregion or super-region), but whose definition resides at another agent in the system. This convention will be followed in the remaining distributed architecture diagrams in this section. In this distributed organization, there is much potential for parallel activity due to the number of planning agents. However, interaction among agents will be high, likewise communication costs, since no planning agent has a complete view of the regions for which it needs to plan.

Another possible scenario involves eliminating the managers and appointing one of the robots as the coordinator of actions over the table. Two possible ways to model this situation are shown graphically in Figure 5.12 and Figure 5.13. Both distributions require five planning agents. In this scenario, there is less potential for parallelization due to fewer planning agents. However, since each planning agent has a more complete view of the regions for which is to plan, agent interaction will be lower and thus communication costs will be less.

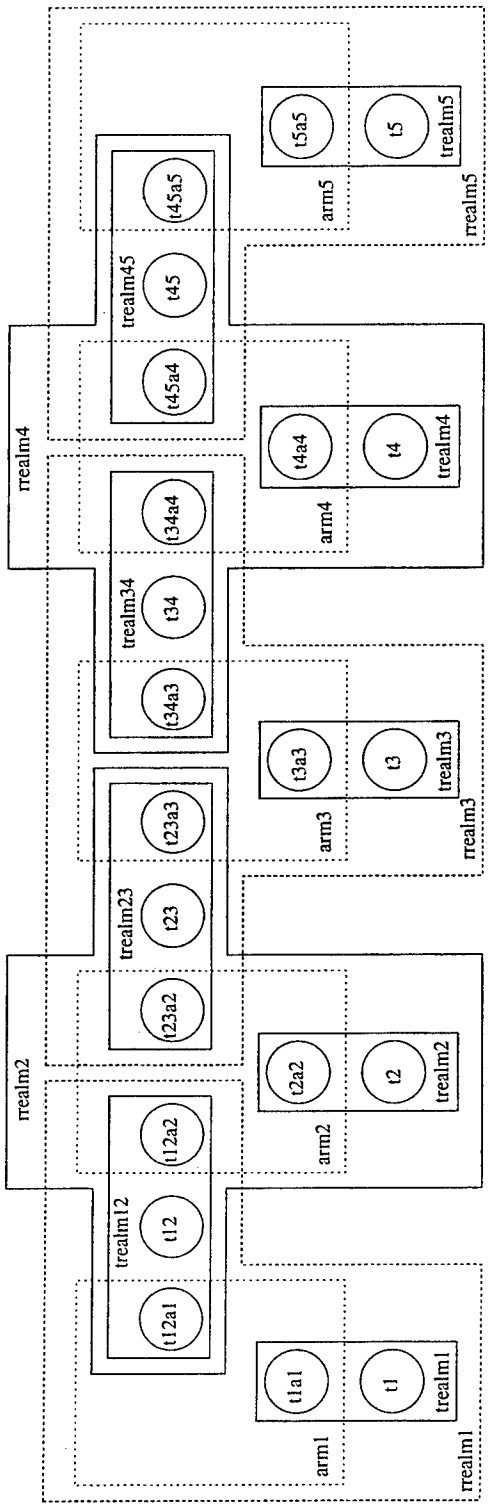


Figure 5.10: Multi-table Blocks World Example Organization

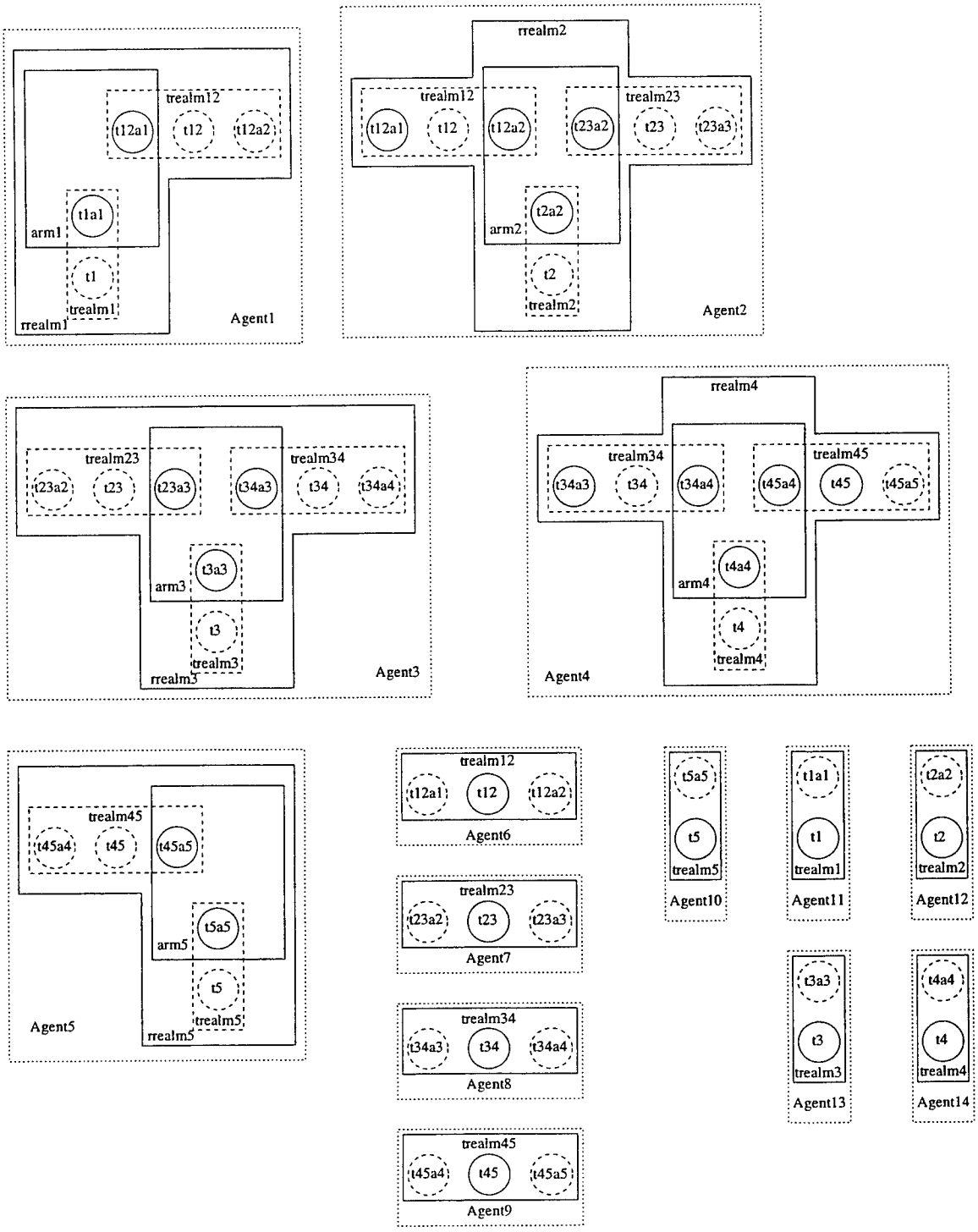


Figure 5.11: A Fourteen Agent Distribution

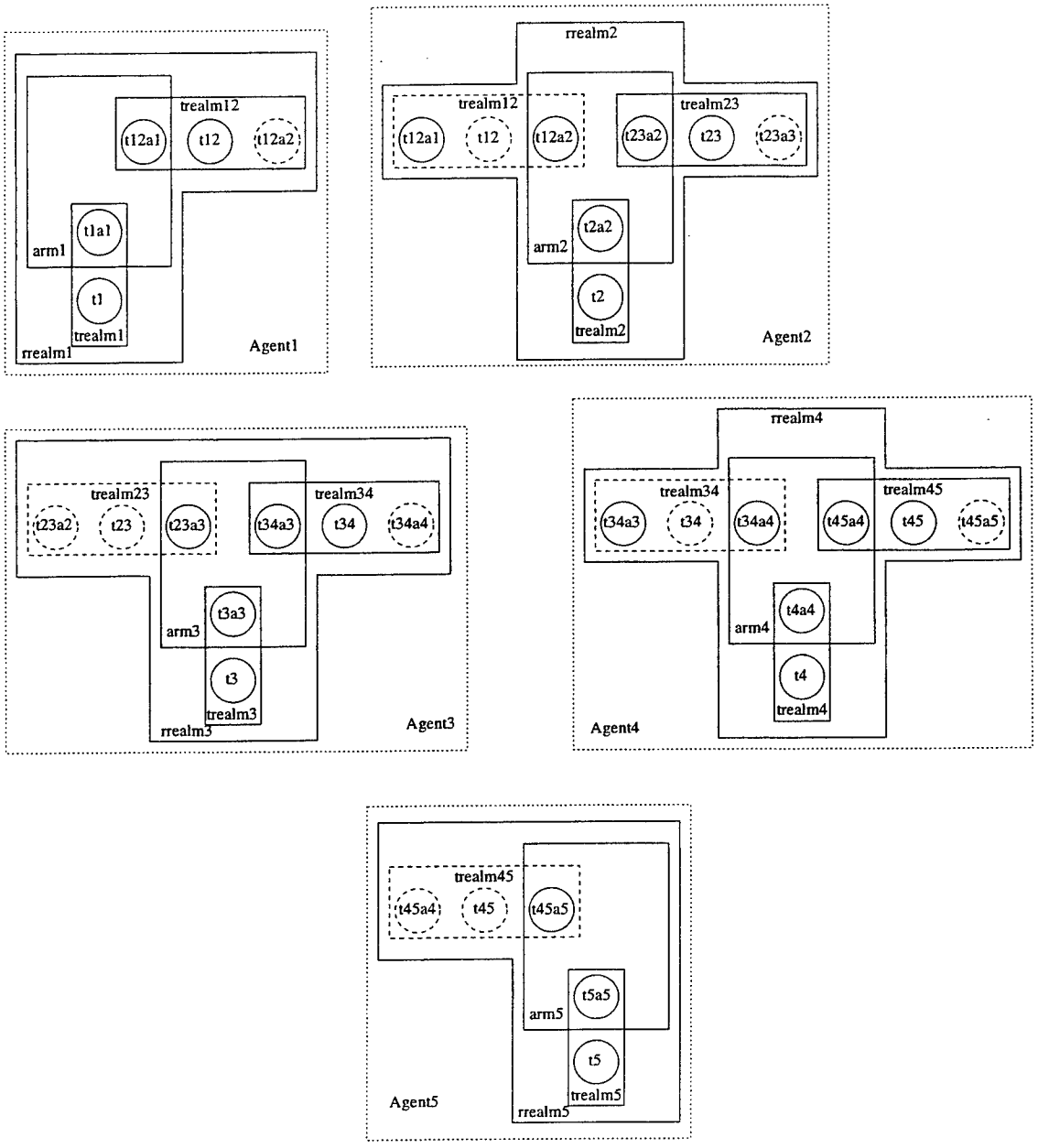


Figure 5.12: A Five Agent Distribution

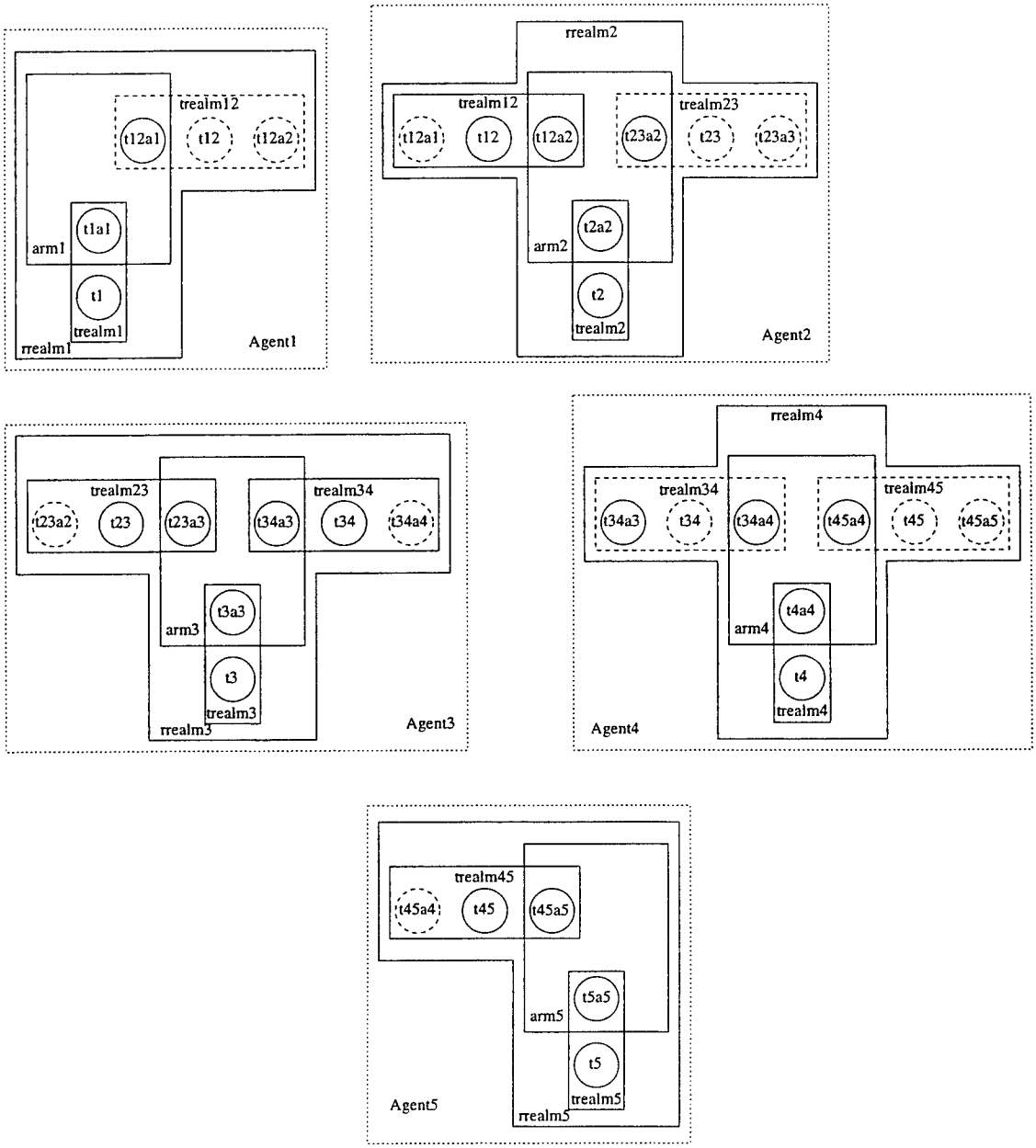


Figure 5.13: Another Five Agent Distribution

We can also model a third situation where again, each robot plans its own actions, but the coordination of the arms is centrally controlled. This distribution requires six DCONSA agents and the organization is shown graphically in Figure 5.14. In this scenario, we can observe the effects of centralized management on planning parallelization and agent interaction.

There are, of course, many more possible distributions. The point to observe is that DCONSA has enough flexibility to model many planning scenarios for a given problem description. These scenarios can involve changing the number of planning agents, or fixing the number of planning agents. In all cases, the planning scenario can be modeled without changing the underlying problem description.

5.5 The Distributed Search for a Plan

In this section, we focus upon the distributed search for a plan. In DCONSA, this search takes the form of a distributed constraint satisfaction problem where constraints and plans are organized in a distributed architecture, as described in section 5.4. Search control is decentralized. As the search process moves from one agent to another, control of the search is passed as well. In addition, to exploit the multiagent power inherent in a distributed system, parts of the search are conducted in parallel. Finally, when two agents interact during the search for a plan, they participate in a negotiation to determine how they will interact.

In the following sections, we discuss how we can integrate decentralized control, parallel searches, and interagent negotiation, while guaranteeing that the planning process proceeds in a coherent manner. This combination of features in a distributed planning system is unique to DCONSA.

5.6 Distributing a Localized Search

In this section, we first describe a slight difference in DCONSA's use of incarnations and then we describe how GEMPLAN's localized search may be distributed. Distributing the localized search requires that we make explicit the record of where the search process has

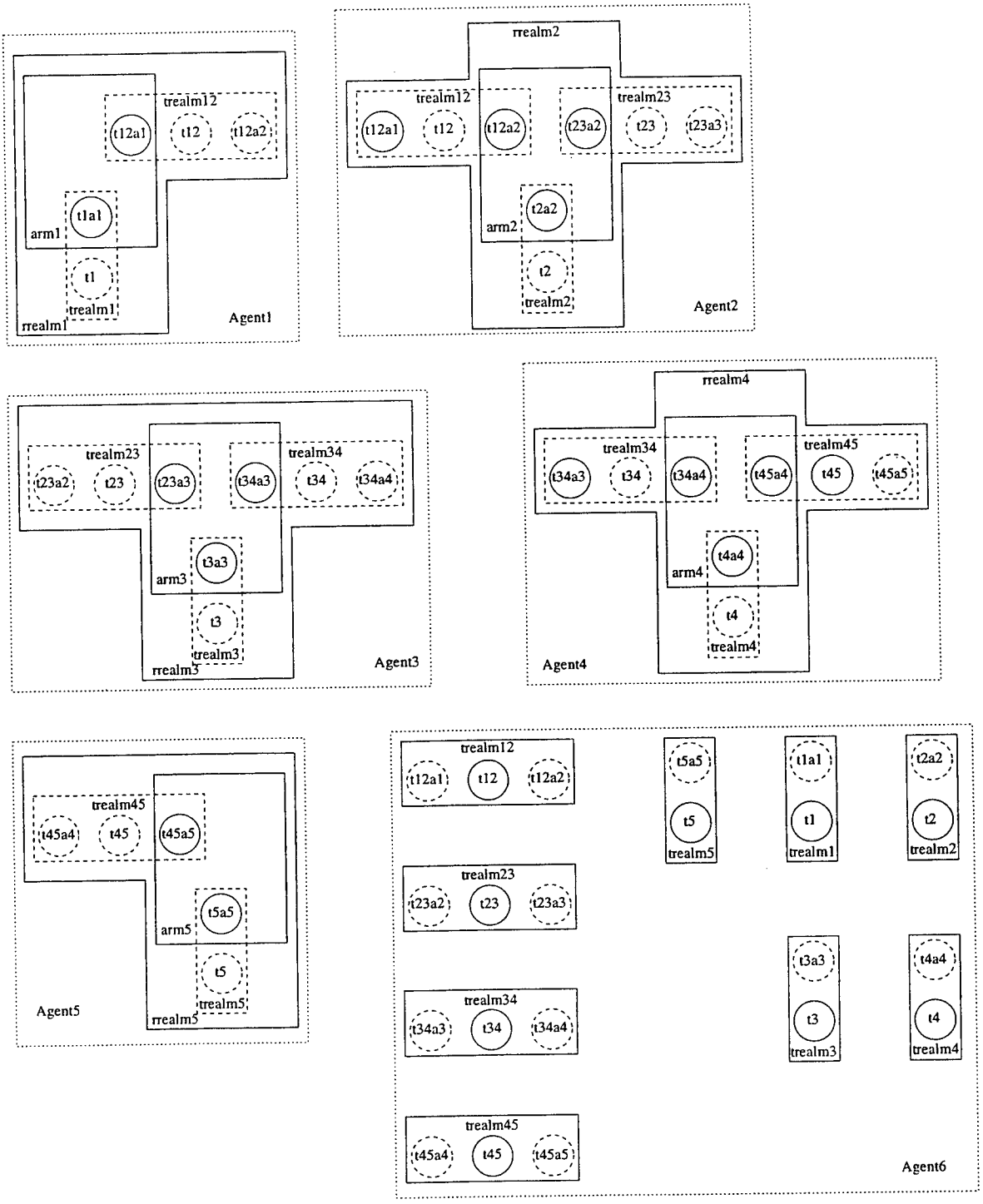


Figure 5.14: A Six Agent Distribution

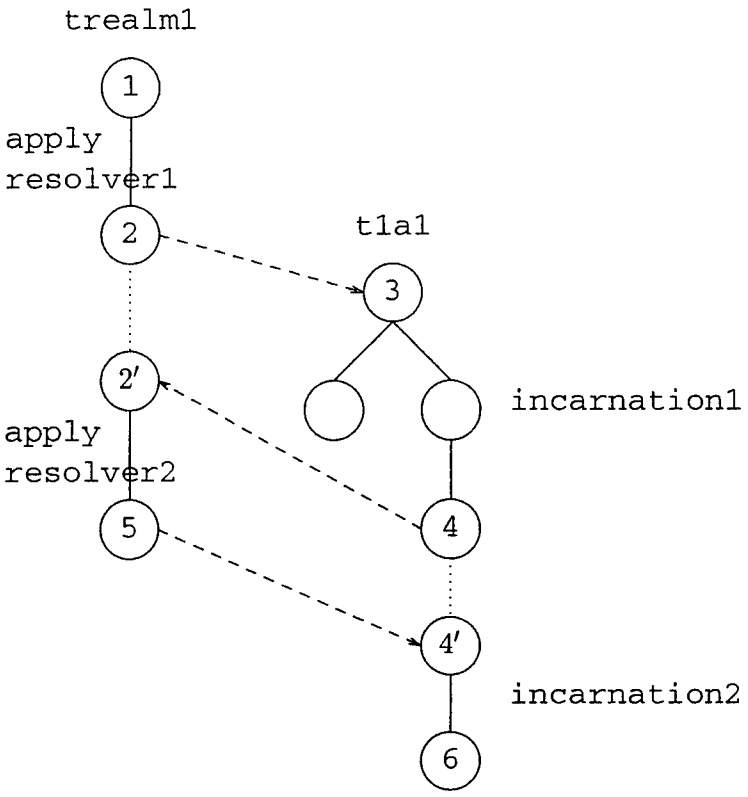


Figure 5.15: Incarnations of Search Space in DCONSA

been and where the search process needs to go. In this section, we present the methods we have developed for conducting this type of localized search in a distributed fashion.

5.6.1 DCONSA's Incarnations

In DCONSA, a new incarnation is created each time that search is conducted in a region. This includes a region search induced by a super-region, as well as a region search that is interleaved with subregion searches. In GEMPLAN, a new incarnation is created only when search is initiated by a super-region. A region search that is interleaved with subregion searches is considered a single incarnation. We have eliminated this distinction so that multiple visits of a search to a region are treated in a uniform fashion. When the search in a region is reincarnated, the first node in the new incarnation is a copy of the last node in the last incarnation. In this way, search state is carried over from one incarnation to the next. When backtracking occurs, it is confined within a single incarnation.

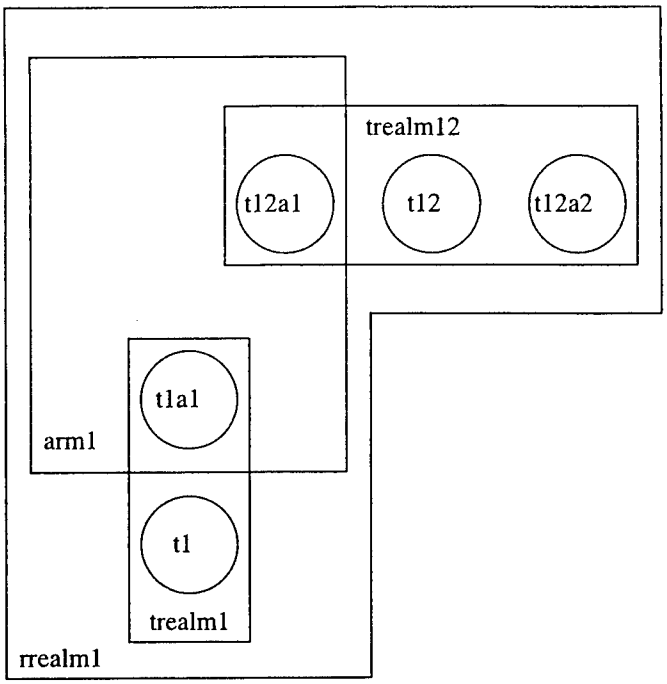


Figure 5.16: Region Organization of RobotRealm *rrealm1*

To illustrate, consider the search spaces presented in Figure 5.15 and the graphical representation of the regional architecture of Figure 5.16. In this example, search begins in TableRealm *trealm1* at node 1. The application of resolver1 modifies the plan of TableArm *t1a1*. Thus search in *trealm1* stops at node 2, and the new plan of *t1a1* is checked with its region constraints. The flow of search between region search spaces is represented by the dashed arrows. This search creates the search space of incarnation1 beginning at node 3 and ending at node 4. The search now returns to *trealm1* at node 2′. Node 2′ is a copy of node 2, that is, they are essentially the same node, represented by the dotted line, except that node 2′ is at the head of a new incarnation. The application of resolver2 modifies the plan of *t1a1*, and thus search in *trealm1* stops again. A new incarnation is created in *t1a1* with node 4′, a copy of node 4, at its head. If the search fails at node 6, the search will back up to node 4′. If the search fails here as well, the search process will stop backtracking within the search space of *t1a1*. The backtracking process will proceed at node 5.

5.6.2 The Distributed Version

In a centralized planner such as GEMPLAN, a record of where the search process has been and where it needs to go can be maintained implicitly in the call stack with a recursive local search function. In a distributed planner such as DCONSA, this record needs to be explicit in order to facilitate transfers among regional search trees which exist in distinct planning agents. In DCONSA, as a search process progresses from region to region, a list of regions whose local constraints must be checked is maintained and associated with that search process. This list is implemented as a stack and serves as a control structure for the search. This structure is termed a *region stack* and each search has a region stack associated with it, guiding its progress through the distributed architecture.

To illustrate, consider a search in the region *arm1* in Figure 5.17. If search in this region results in the addition of an event, say $Pick(a)$, in the subregion *t1a1* , then search in *arm1* is suspended so that the affected region, *t1a1*, can be searched. *Arm1* is pushed onto the region stack and then *t1a1* is pushed onto the region stack. In this way, search will eventually return to *arm1*.

Once the search in *t1a1* is complete, all super-regions of *t1a1* are pushed onto the region stack since the addition of the event $Pick(a)$ has changed their plans as well. In this instance, *rrealm1* is pushed onto the the stack (we need to ensure that a is within reach), and *trealm1* are pushed on the stack (we need to ensure that a is clear before it is picked up). Thus, the search process is guaranteed to move through all regions which could be affected. To limit the potential size of the region stack, a region is only pushed onto the stack if it is not already present. [12] This bounds the length of the region stack by the number of regions in the system.

The region stack we have described controls inter-region coordination during search in the forward direction, but it does not aid backtracking in the case of a regional search failure. Recall that a distributed search process can be viewed as a series of regional searches in regional search trees. One way to direct backtracking would be to associate a second region stack with each search process to record the list of regions where that search has been. One drawback of this method is that the length of this "backtrack" stack can not be bounded in the same manner as our original region stack. For this reason, we employ a different method to direct backtracking. Each agent maintains a variable, *region-transfers*, which is a list of "ties". A tie is a data structure of the form:

[12] Thus, since *arm1* was already on the region stack, it was not pushed on with *t1a1's* other encompassing regions in the previous example.

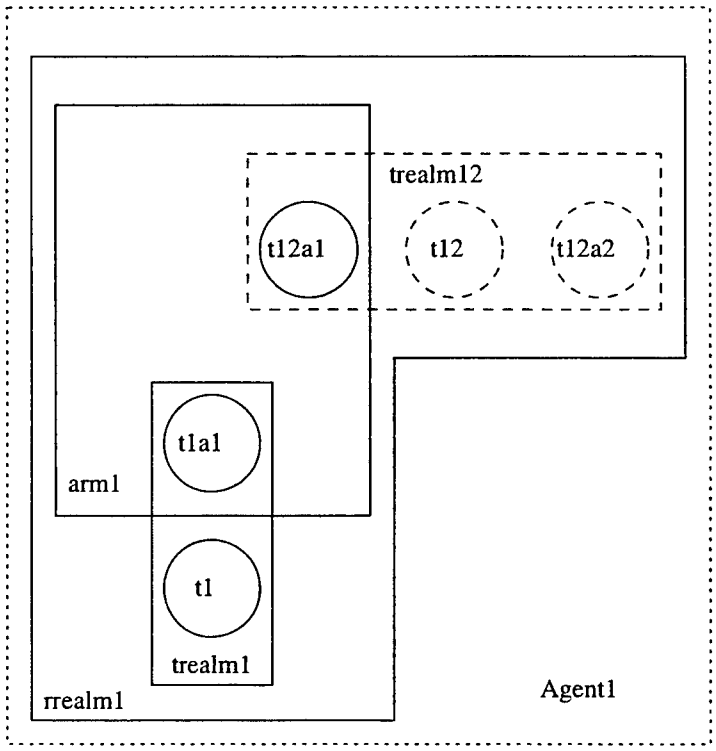


Figure 5.17: Example Region with Nonlocally Defined Subregions

(region-id . incarnation-node).

The region-id indicates the region which was searched immediately before the portion of the search space in the agent headed by the node incarnation-node. Thus, if a search backtracks to the beginning of an incarnation, the agent can look up the tie associated with this incarnation and determine the region where search needs to backtrack. When search backtracks to this region, the last node visited in its most recent incarnation is revisited.

One other consequence of distributing GEMPLAN's localized search strategy is that regions may no longer have direct access to the plans of their subregions. This occurs when some of a region's subregions reside at other agents in the system. As noted previously, changes that are made in the local plan of a region, must be reflected in that region's encompassing regions before search is carried out in those regions. For example consider the RobotRealm *rrealm1* in Figure 5.17. When changes occur in the region plans of any of *rrealm1's* subregions, those modifications must be reflected in the plan of RobotRealm *rrealm1* before *rrealm1's* search returns to RobotRealm *rrealm1*. This includes modifications to the plans of the nonlocally defined subregions TableRealm *trealm1* and TableArm *t1a1*.

One possible strategy for propagating any changes is to update the plan in a region each time the search in one of its subregions completes. Another strategy is to update a region's information about each of its subregions' plans just before the region is searched. The second strategy has the advantage that when multiple subregions of the same region have local plan changes, the region is only updated once. Because this potentially saves message traffic within the system, we have implemented this strategy in DCONSA.

5.7 Exploiting Multiprocessor Power

DCONSA, as it has been described thus far, makes it possible to model interaction among executing agents as well as interaction among agents involved in the *distributed process* of plan development. So far however, we have only described distribution of the localized search of GEMPLAN. Thus, the process that has been described involves the serial search of region search spaces which are distributed among planning agents. It would be advantageous to exploit the inherent capability of a multiagent planning system to conduct the search in parallel. In this section we describe a distributed parallel search strategy that employs a two stage planning process.

5.7.1 A Two Stage Approach

The basic strategy for exploiting the multiprocessor power of our multiagent planning system is to plan for conjunctive goals in parallel and then to combine these plans into one coherent plan. This is accomplished by partitioning the regional constraints of each region into two sets - constraints that pertain to the development of a plan for a single goal (goal constraints), and those constraints that deal specifically with the interaction of plans for conjunctive goals (interaction constraints). As an example of an interaction constraint, consider two goals to configure blocks: on(a,b) and on(b,c). The plan $pick(a) \rightsquigarrow put(a,b)$ will achieve the first goal and the plan $pick(b) \rightsquigarrow put(b,c)$ will achieve the second goal. However, since both plans involve the movement of block b, events in these plans must be timed properly to create a coherent overall plan. The regional constraint which ensures that stacking events for these conjunctive goals occur in a logical order is an example of an interaction constraint. [13] In addition to partitioning the regional constraints, each regional search space will also be partitioned. A distinct search space is created each time a region plans for a new goal. Thus, a region will maintain a search space relative to each goal for which it has created a local plan. This may or may not include every conjunctive goal in the system. Furthermore, a separate search space will be created for the combination of these local plans. To illustrate, consider Figure 5.18 which depicts an agent, a1, which plans for three regions, r1, r2 and r3. Each of these regions has a distinct search space pertaining to each goal for which it has constructed a local plan, and a distinct search space for the combination of these plans.

In the first stage of planning, each agent which has a region with a goal will initially assume responsibility for developing a plan for that goal. Each of these searches will proceed in parallel as described previously, except that regional interaction constraints will be ignored. In addition, events and event relations will be associated with the goal that they are intended to achieve. Thus, when an agent plans for a goal in a particular region, it will use the regional search space created for that goal and check constraints using only those events and relations which are locally associated with that goal. Once a plan for each conjunctive goal is found, the agents move into the second stage of planning.

During the second stage of planning, each region combines the local plans it has constructed thus far. This is accomplished by a serial search of each region's "interaction" space. During this search, the only constraints which are considered are those pertaining to the interaction of plans for conjunctive goals. These constraints are tested using the events and event relations of every local plan. When the last of the stage one searches completes, the agent which was directing this search consults with the other system agents

[13] This constraint appears in [46], pg 46.

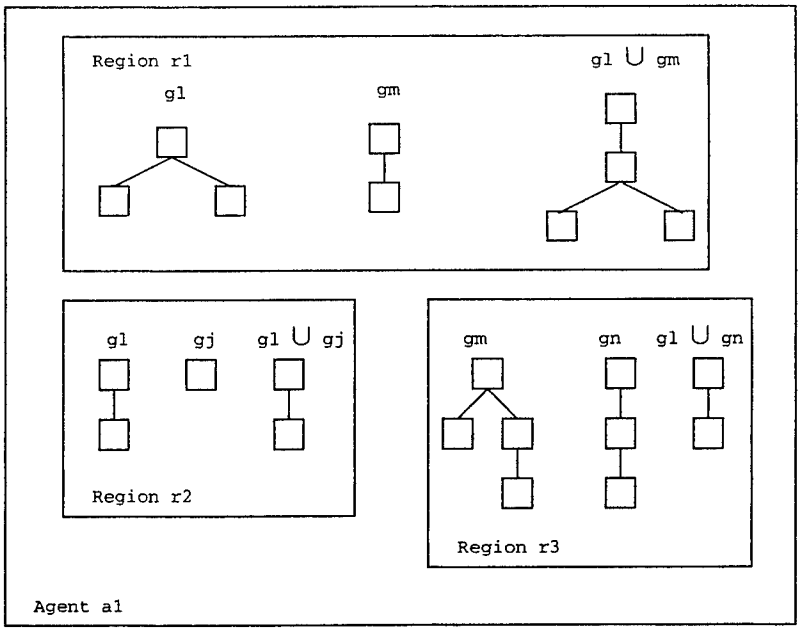


Figure 5.18: Conceptual View of Partitioned Search Spaces in an Agent

to initialize the region stack for the interaction search. This region stack initially contains every region which was given a goal at the time of that region's initialization. The agent which constructs this region stack finds out which agent plans for the first region on the stack and initiates the serial interaction search.

Interactions among plans for distinct goals should be resolved at the lowest level regions first and from there, the interaction search should continue up the regional architecture. To achieve this, DCONSA permits the interaction search to occur in a region only if the interaction search has already successfully completed in all of that region's subregions. Before a region is searched in the interaction space, this condition is checked. If the interaction search has not been completed at some number of subregions, the region is pushed onto the list of regions to check, and then the appropriate subregions are pushed onto this list as well. This gives the interaction search the desired form.

If the search in the interaction space is successful, then a plan which achieves each of the conjunctive goals will have been created. A search failure in the interaction space can have different consequences depending upon the characteristics of the domain being considered. If there is only one possible plan to achieve each of the conjunctive goals, then a failure in the interaction search space indicates that no plan exists which will solve the complete set of conjunctive goals. If at least one of the conjunctive goals has more than one possible

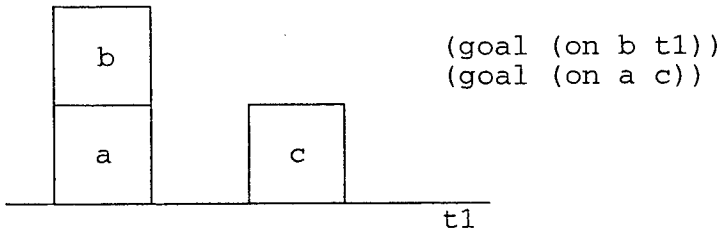


Figure 5.19: An Example Leading to Redundant Events

plan, then the search space should be revisited using a different combination of plans.

5.7.2 Working Out Some Details

One consequence of planning for conjunctive goals in parallel is that redundant events can be created. For example, consider the goals and configuration of blocks shown in Figure 5.19. Planning for these goals in parallel will result in two plans: $pick(ev1, b) \rightsquigarrow put(ev2, b, t1)$ and $pick(ev3, b) \rightsquigarrow put(ev4, b, t1) \Rightarrow pick(ev5, a) \rightsquigarrow put(ev6, a, c)$. [14]Obviously, when these two plans are combined into one overall plan, the redundant pick and put events using b should be merged. The first time an element region is visited in the interaction search, the redundant events in that region are merged. A set of events is redundant if all of the events are the same event type, their parameters are identical (except for their ids), and if they are unordered with respect to one another. This last condition ensures that no temporal relations are violated when these events are merged. To merge these events, a new event is created using the parameters of the redundant events and the id is set to be the concatenation of the redundant event ids. This new event is substituted in every event relation involving one of the redundant events. This new set of events and event relations is set as the initial plan in the element's interaction search space. The first time a group region is visited in the interaction search space, its event relations are updated to reflect any redundant event merging that may have occurred in its subregions.

Mechanisms for the partitioning of regional constraints into goal constraints and interaction constraints need to be formalized. At this stage, the process of partitioning regional constraints is guided by intuition and corrected later by experiment. We also intend to consider ways of classifying interactions so that we can investigate whether it is possible to conduct a distinct interaction search. There may be domain specific characteristics that

[14]Every event is identified by a unique id so that distinct events with identical parameters may be differentiated.

make it impossible to decompose the search into these two phases. We have found some promising ideas in the work of Yang, Nau, and Hendler [92].

5.8 Incorporating Agent Autonomy

The previous sections discussed the distribution and partial parallelization of a localized search process. We now turn our attention to modeling agent autonomy within the context of distributed localized search. As described in the previous section, in DCONSA the distributed localized search is carried out at a regional level. We incorporate agent autonomy by adding a second search tier at the agent level. At this second level, agents use negotiated forms of interaction to guide search at the regional level.

5.8.1 A Two Tiered Approach to Search

For agents to be semi-autonomous, they must have some control over their own actions. In DCONSA, autonomy comes through an agent's ability to dynamically determine how it will interact with other agents during the distributed planning process. This distributed process involves a search that progresses forward as well as backtracks. Agents exercise their autonomy by selecting from one of three modes of interaction during planning. This selection is made each time interaction is required between agents and is not tied to past choices. The three modes of interaction are:

1. (Mode I) An agent may choose to assume responsibility for plan development. [15] Should this occur, that agent will direct the search for a plan for a particular goal until a plan is found or until it can find another agent that will take over this plan responsibility.
2. (Mode II) An agent may perform a portion of the search locally, and then pass the result of this search to the agent that is directing the search.
3. (Mode III) An agent may send the local information required to continue the search to the agent directing the search. This allows the directing agent to continue the search on its own.

[15] The search for a plan for a given goal is always the responsibility of some agent in the system.

To incorporate these three alternative modes of interaction in a distributed localized search, one can think of the search process as being two tiered, with an agent search layer built upon the distributed localized search layer. The autonomy of the agents resides at the agent search layer and it is this layer which directs the distributed localized search using a negotiated interaction mode as a guide. The following paragraphs describe the interplay between these two layers.

Mode I - When an agent chooses to take over plan responsibility for a goal, it directs the search in the localized search layer using the region stack to control where search should progress. In the course of this search, it is possible that a region will be encountered on the region stack whose definition is unknown to the agent. At this point, the agent knows it must request assistance from whatever agent has the definition of the required region. The requesting agent can ask the assisting agent to interact in any of the three modes. If no preference for interaction between these two regions has been indicated previously, the default is to try each mode in succession until one is acceptable to the assisting agent. The order in which these requests are made is as follows: first Mode I - the assisting agent is asked to assume plan responsibility, then Mode II - the assisting agent is requested to plan locally and return a result, and as a last resort Mode III - the assisting agent is asked to pass the definition of the required region to the requesting agent.

Mode II - When an assisting agent agrees to aid a requesting agent in developing a plan, the assisting agent is told which region the requesting agent wants it to search. This is accomplished by pushing a single item, the desired region, onto the region stack associated with the request. Using this region stack, the assisting agent conducts a localized search in that region and determines what other regions need to be searched as a result of that particular localized search. The result returned by the assisting agent includes an indication of whether the search completed successfully and a list of regions that should be added to the region stack for the search.

Mode III - If an assisting agent receives a request to send a region definition, the assisting agent must send a description of the requisite region including that region's local constraints and resolvers as well as a description of the progress made within the search space associated with that region. In addition, the ties which link that search space to other regional search spaces must be transmitted. The region definition is then removed from the assisting agent's records, along with any existing pointers to the search space that was sent. Thus, only one definition of each region exists in the system.

We have restricted our attention to cases in which there is only one definition of each region for the sake of simplicity. With only one instance of a region in the system, all

planning for that region must be accomplished by an agent with a complete view of that region. If more than one instance of a region existed in the system, additional coordination would be required to guarantee that the agents holding the multiple instances had consistent views of planning state. Such a scheme may be desirable in situations where redundancy is a requirement.

5.8.2 Negotiated Interaction

To make effective use of agent autonomy, agents must have a basis for selecting one mode of interaction over another that produces desirable behavior. In a cooperative system, each agent should attempt to strike a balance between its self interest and its interference with other agents. Whenever one agent interacts with another, it necessarily interferes with what the other agent was doing by requiring processing of its request. Of course, the degree of interference depends upon the type of request or interaction. Thus, our interaction modes can be viewed as representing different levels of interference. Clearly Mode I requires the most interference, while Mode III requires minimal interference and Mode II is somewhere in between. By incorporating autonomy, agents should be able to make decisions about how to interact that tend to minimize interference in the system as a whole. To achieve this desired behavior, we use a heuristic decision criterion that is based upon relative agent work load. By attempting to dynamically balance agent work load, we try to reduce agent interference.

When a requesting agent asks an assisting agent for aid in planning using a particular interaction mode, the request carries with it a measure of what the requesting agent's work load would be if the request were accepted, wl_{req}. In making its decision whether to accept the request, the assisting agent projects what its work load would be if it accepted the request, wl_{assist} and compares this to wl_{req}. The request is accepted if:

$$wl_{req} \geq K_m \times wl_{assist},$$

where K_m is an acceptance threshold for the interaction mode m. Different acceptance thresholds for Mode I and Mode II may be defined. Since the agents are cooperative, and a request to use Mode III is the last available option, Mode III requests are always accepted.

This acceptance criterion can be viewed graphically as shown in Figure 5.20. With wl_{req} plotted on the vertical axis, and wl_{assist} plotted on the horizontal axis, the assisting agent

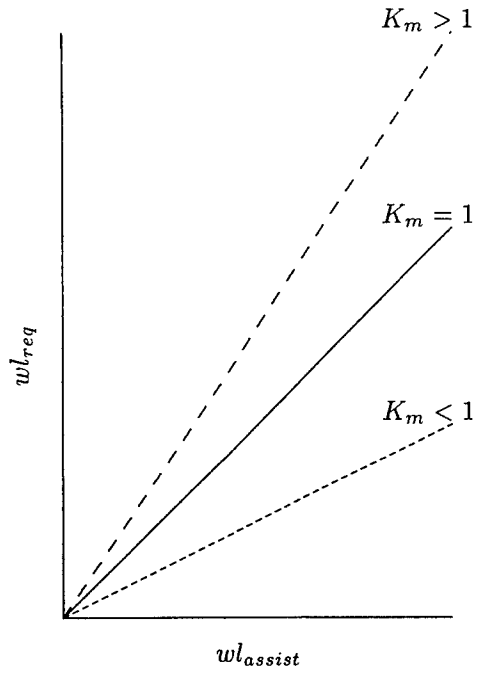


Figure 5.20: A Graphical Representation of the Acceptance Criterion

will accept the request whenever the coordinate (wl_{assist},wl_{req}) lies on or above the line of the above equation. The slope of the line representing the acceptance criterion can be shifted by modifying the acceptance threshold for a given mode. Three examples of this line are shown: one for $K_{mode} = 1$, one representing a shift when $K_{mode} \geq 1$, and one representing a shift when $K_{mode} \leq 1$.

The success of this acceptance criterion will necessarily depend upon the accuracy with which we can assess an agent's work load, wl_{agent}. To assess this value, we have focused upon two parameters: committed work, cw_{agent}, and current responsibilities, cr_{agent}. Committed work refers to work required to fulfill requests for planning aid which an agent has already agreed to perform. The other measure, current responsibilities, refers to the work required to conduct searches for plans for which the agent currently has plan responsibility. The combined assessment is a weighted sum of these two measures:

$$wl_{agent} = C_{cw} \times cw_{agent} + C_{cr} \times cr_{agent},$$

where C_{cw} and C_{cr} are parameters used for fine tuning. The measures cw_{agent} and cr_{agent} are given values by approximating the number of region searches that the agent will complete

to accomplish its committed work and current responsibilities respectively.

To give a value to cw_{agent}, we use a heuristic that looks at planning being done for other agents that is in progress, but temporarily suspended, and requests for planning aid to which the agent has committed, but not yet started:

$$cw_{agent} = \text{suspended aid} + \text{unstarted aid}.$$

Searches being conducted for other agents using Mode II can be suspended for only one reason - to gather information about subregion plans before planning in the given region takes place. We use *region locks*, a device analogous to process locks, to guarantee that the given region does not move from the agent while the search is suspended. [16] Therefore, the agent is guaranteed to eventually search this region. This component of cw_{agent} is the number of suspended Mode II searches.

To measure the second component of cw_{agent}, we look at the agenda of requests for planning aid that the agent has committed itself to fulfill and approximate the number of region searches required by these requests. Each request will involve conducting a search using either Mode I or Mode II. Also, each request will have an associated region stack used to guide search. By committing to fulfill a request, an agent will, at a minimum, conduct a search in the region at the top of the region stack. The only exception to this rule occurs if at the time that the request is processed, it is discovered that the definition of the region in question has been transmitted to another agent in the system. In this situation, no search would take place, and the requesting agent is notified that the region has moved. This component of cw_{agent} is the number of requests on the agent's agenda.

To determine a value for cr_{agent}, we examine the suspended searches for which an agent currently has plan responsibility. These Mode I searches can be suspended for two reasons: to gather subregion plan information before a region search, and to wait for a reply from a Mode II request. The queries associated with each of these cases contain information to restart the suspended search. In particular, they will hold the region stack for the suspended search. Each region on the region stack of a suspended search that is locally defined will eventually be searched by the agent with plan responsibility. This is true unless that region moves in the interim to another agent in order to balance the planning workload. Thus, to determine an estimate of the work involved in completing these suspended searches, cr_{agent},

[16] These are discussed in Section 5.8.3.

we determine the number of locally defined regions in the region stack of each suspended Mode I search.

One result of giving agents autonomy, specifically the option of transmitting region definitions, is that the distribution of region definitions is dynamic. It is possible for an agent's knowledge regarding which agent plans for a particular region to become outdated. As a result, queries regarding the update of subregion plans and requests for interaction in plan development may be sent to inappropriate agents. In DCONSA, if an agent receives a reply indicating that the region involved in its query is unknown to the receiving agent, a broadcast message is sent to locate the agent that currently plans for the given region. Associated with this broadcast query is a function and a set of arguments to call when the appropriate agent is found. The call to this function creates a new query and directs it to the appropriate agent.

5.8.3 Distributed System Issues

In this section, we discuss two issues regarding agent autonomy in the distributed search - search completeness and coherency. For instance, is it possible for planning not to progress because agents get in a loop of asking someone else to plan for them? Or, what happens when an agent is conducting a search in a region for one goal, and some other agent wants that region sent to it to continue a search for a different goal? In the following paragraphs, we discuss the mechanisms we provide for dealing with these issues.

First, we'll look at search completeness. We call DCONSA's planning agents *semi-autonomous* because they work cooperatively together during the planning process. That is, an agent that has been asked for planning aid, must agree to one of the planning modes. No agent is allowed to refuse to help another agent. Therefore, agents are not completely autonomous, because they are forced into one of three interaction modes.

This forced cooperative behavior coupled with the notion of plan responsibility, prevents a situation such as the one described above. Once an agent has plan responsibility, it can request another agent to take over that plan responsibility. However, if that agent refuses, the first agent retains plan responsibility. The first agent can then ask for another agent to do the planning, but if this is refused as well, then the other agent must send the required planning information to the agent with planning responsibility. When the region is received, the requesting agent is guaranteed to conduct a search in that region.

Now, let us consider the situation where agents get into the following pattern of passing plan responsibility:

1. An agent accepts plan responsibility for a goal, g1, because it holds the definition of the next region, r1, to be searched for a plan for g1.
2. The agent puts the request for that search on its agenda.
3. Eventually, the agent takes the request off the agenda only to discover that r1 has since been transferred to another agent as the result of search for a plan for a goal other than g1.
4. The agent requests the agent with the r1 to accept plan responsibility for g1.
5. Repeat

Although a loop of passing planning responsibility such as this is possible, it does not mean progress is not being made. Consider that every time r1 moves from one agent to another, a search in r1 for some goal is conducted. Thus progress is being made and eventually the searches for goals other than g1 will progress to the point where they either complete, or no longer need to search r1. At this point, either the agent that has the definition of r1 or the agent that currently has planning responsibility for g1 will complete the search for g1 in r1.

We now turn our attention to search coherency. One mode of interaction among agents involves the transfer of a region definition between two agents. Furthermore, the first stage of the search process involves multiple parallel searches. Thus, it is possible that an agent may be currently conducting a search in a region for one goal, when it receives a request to send that region definition to another agent. This request would result from a search for a different goal. As explained earlier, only one definition of each region exists in the system. Therefore, for search coherency, we would like the agent to finish its current search in the region before shipping it to another agent.

To achieve this desired behavior, we use a device called a *region lock*. A region lock is conceptually similar to a process lock. Whenever an agent begins a search for a goal in a region, the agent places a lock on that region associated with that goal. When the search for that goal in that region completes, the lock for that region associated with that goal is unlocked. Thus a region is locked, if there is a lock for any goal on that region. Likewise, a region is unlocked if there are no locks for any goal on the region. Note that for any given goal, there is at most one region locked for that goal at any time. When a request to ship

a region is received, the agent checks to see if the region is locked. If it is, the request is put onto the agent's agenda and is periodically retried until the region becomes unlocked. Note however, that locking a region does not prevent requests for information about that region from being serviced. For example requests for information regarding plans within that region can be answered whether or not the region is locked. The lock only prevents the transfer of the region between agents, not the transfer of information about the region.

The only way agents could become deadlocked, is if an agent had a lock on a region and continued to hold that lock until one or more locks on a region in another agent were released. If this situation were possible, then we could "mirror" the scenario thus deadlocking two agents.

We have designed DCONSA so that this will not occur. An agent will unlock a region involved in a search before it requests the transfer of another region for that search. An agent may suspend a search in a locked region while waiting for subregion plan information and lock another region while it works on another goal. But, as pointed out above, the subregion plan information will not be blocked by other locks. Thus the suspended search in a locked region can progress regardless of locks on other regions. Therefore, deadlock can not occur.

Consider a simple case, as shown in Figure 5.21, where Agent A has a lock for goal g1 on region r1 and Agent B has three locks for goals g2, g3, and g4 respectively on region r2. Let us assume that Agent B completes its search in r2 for g2 and therefore releases that lock. Furthermore, let us assume that the next region to be considered in the search for g2 is r1 and that after a brief negotiation, the agents mutually agree that Agent A will transfer r1 to Agent B. Agent B now must wait for Agent A to release its lock for g1 on r1. Note however, that the search resulting in the lock for g1 can not be blocked by any of the locks on r2. The search for g1 in r1 may be suspended to gather plan information from r2, but this is not prohibited by the locks. The only behavior prohibited by the locks on r2 are its transfer to Agent A for the search of a plan for g1. But the search of a plan for g1 will not progress to r2 without first unlocking r1. Thus, there is no way that the locks on r2 can prevent the search for g1 in r1 from completing. Eventually Agent A will release its lock on r1 and the transfer of r1 to Agent B will occur.

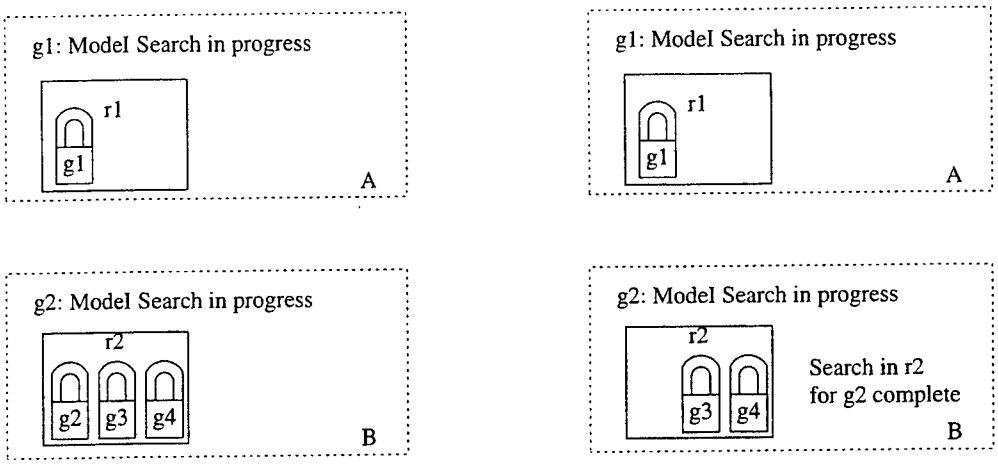


Figure 5.21: Impossibility of Deadlock

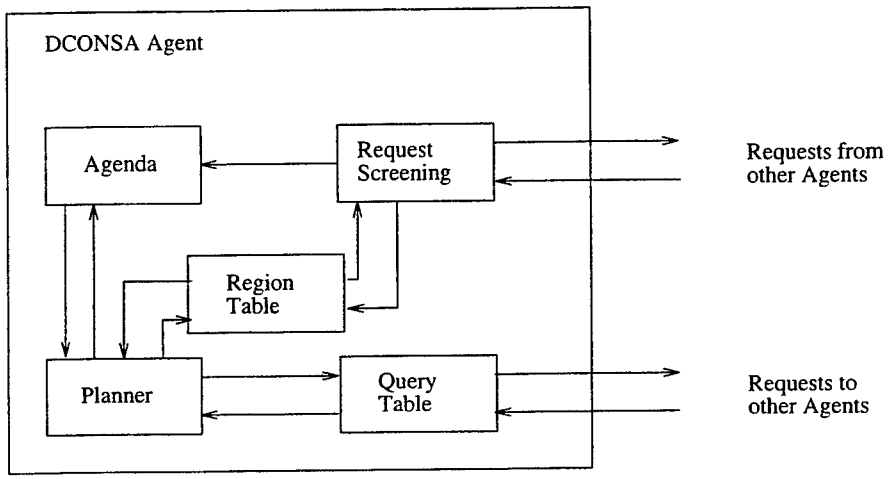


Figure 5.22: Data Flow in a DCONSA Planning Agent

5.8.4 Internal Agent Implementation

In this section, we discuss the flow of data within a DCONSA planning agent and the internal agent control. The data flow within an agent is shown in Figure 5.22. Each DCONSA planning agent has a Planner, a Request Screening component, an Agenda, a Region Table, and a Query Table.

The Planner contains the code that performs the actual search for plans for goals. It can

read and modify the Region Table, the Agenda, and the Query Table.

The Request Screening component processes requests from other DCONSA planning agents. It can place requests for planning aid onto the Agenda, it can reject requests using the acceptance criterion described in Section 5.8.2, and it can process requests for plan and synchronization information.

The Query Table is used to record queries made to other agents. These queries include requests for planning aid, requests for region plan information, and synchronization queries such as termination of the first stage of planning. Replies to these queries are stored here as well.

The Agenda is used to store tasks which the Planner needs to address, namely requests for planning aid. The Agenda can be modified by both the Planner and the Request Screening Component. The Planner may pop items off the Agenda and as well may put things back on the Agenda if action on the item must be delayed.

The Region Table stores the region definitions known to the agent. The Region Table can be modified by both the Planner and the Request Screening component. The Request Screening component may remove a region definition as the result of a ModeIII interaction or it may add this request to the Agenda if the requisite region is locked. The Planner may add and delete region definitions during search as the result of interaction ModeIII.

To complete this discussion of the internal agent implementation, we now consider the internal control of the Planner and the Request Screening component. Diagrams of the internal control are shown in Figure 5.23 and Figure 5.24.

The Request Screening component waits for the arrival of a request from another planning agent. Once a request arrives, it reads the request and determines whether the request is for information or a request for planning aid. If the request is for information, a reply is sent. If the request if for planning aid, the Request Screening component checks the acceptance criterion. If the request is accepted, it is placed on the Agenda, otherwise a reply is sent rejecting the request.

The Planner is driven by tasks on the Agenda and replies to entries in the Query Table. The Planner waits for such an occurrence and processes it accordingly.

This completes our discussion of the distributed search process of DCONSA and our description of how agent autonomy can be exercised. In the following chapter, we describe

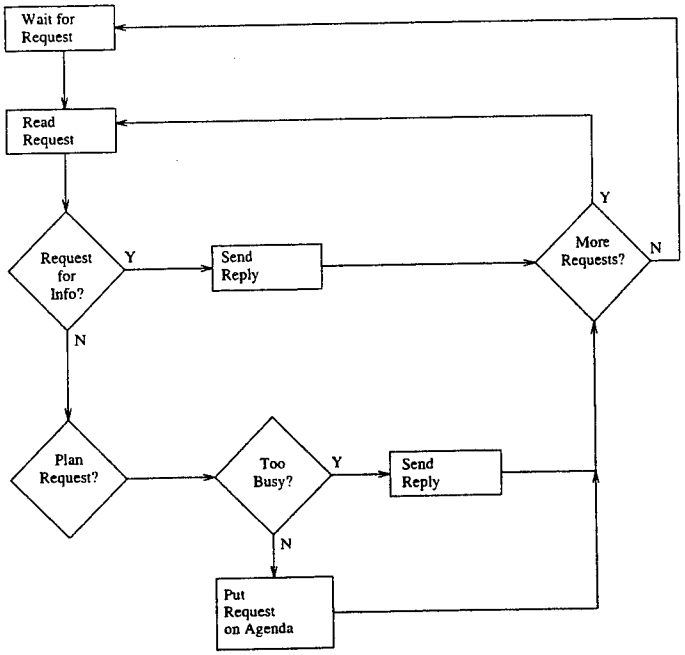


Figure 5.23: Internal Control of Request Screening

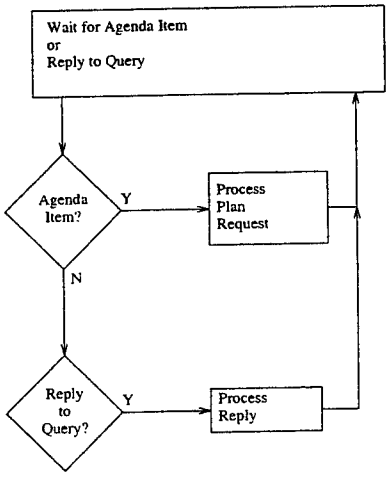


Figure 5.24: Internal Control of Planner

our experiments and discuss their results.

5.9 Experiments

We have run a number of experiments using DCONSA. In this section, we outline the experiments we have conducted and discuss their results. Our experimentation shows that our heuristic decision criterion, based upon relative agent work load, improves planning performance.

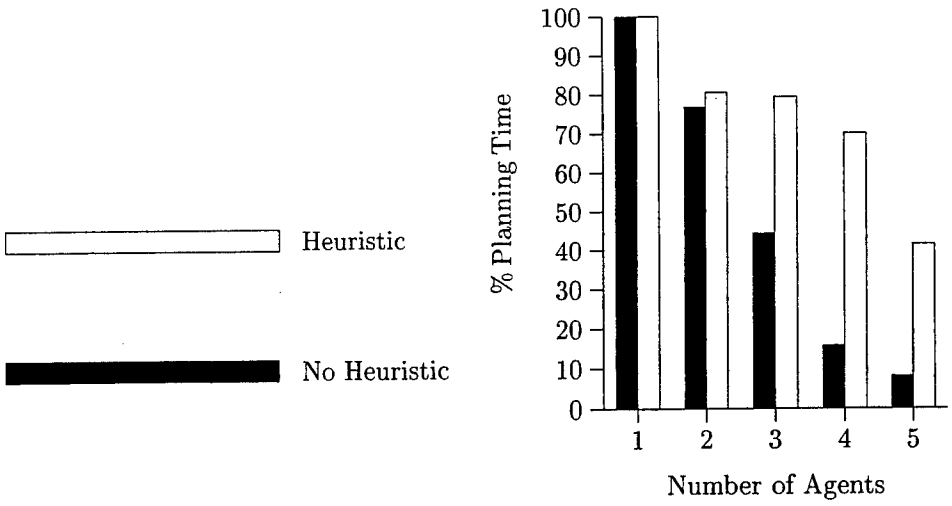
We have conducted our experiments using the instance of the Multi-table Blocks World, described in Section 5.4.2, that involves five arms and nine table surfaces. Each arm has a table surface that it alone can access, and the remaining four table surfaces are each shared between two agents. Thus we are able to model independent action as well as actions which require coordination.

5.9.1 Fine Tuning the Acceptance Criterion

Our first set of experiments focused upon measuring the effectiveness of our heuristic decision criterion for selecting modes of interaction. An effective criterion should result in the ability of DCONSA planning agents to:

- balance agents' individual responsibility for conducting portions of the planning process,
- shift region definitions to appropriate agents,
- and accept responsibility for planning in such a way that the overall process is cooperative, coherent and minimizes agent interference.

Three performance measures indicate how well our heuristic decision criterion enables the DCONSA planning agents to meet the above objectives. These three indicators are agent utilization, overall time for plan construction, and the redistribution of regions to appropriate agents during planning. The latter two indicators are easy to measure and express. Agent utilization is slightly more complicated.



(a) Legend

(b) Agent Utilization in a typical case.

Figure 5.25: Example Agent Utilization Bar Graph

We express agent utilization using a bar graph with %Planning Time measured on the vertical axis and Number of Agents measured on the horizontal axis. Figure 5.25 is a typical example. Each bar represents the percentage of planning time that a particular number of agents were planning in parallel. The black bars represent the case in which agents were given no capability to select among interaction modes [17], thus no decision criterion was employed. The white bars represent the case in which agents use our heuristic decision criterion to make decisions regarding agent interaction. Planning time refers to the simulation time during which at least one planning agent was running. This eliminates measurements of simulation overhead. To compare agent utilization in different simulation runs, where total planning time may differ, we look at percentage of planning time rather than absolute planning time. Therefore, by definition, we will see the utilization of one planning agent at 100% in all agent utility graphs.

In our first set of experiments, we looked at a problem with five planning agents and five goals to achieve. The search spaces associated with the goals are disjoint, that is, each search travels through a unique set of regions. Since the searches will never try to pass through the same region, we call the searches *non-overlapping*. Because there are five non-overlapping searches and there are five planning agents, it follows that we should expect a high level of agent utilization to be possible.

[17] Agents always propose and accept interaction Mode 1

Furthermore, we used this first set of experiments to compare two different heuristic decision criteria. The acceptance criterion described in Section 5.8.2 is actually the second criterion we tried. The old criterion was a rougher estimate of an agent's work load. This estimate was merely a sum of the number of goals for which the agent currently has plan responsibility and the number of accepted requests for planning aid. Our new heuristic, you will recall, attempts to measure the number of regions that will be searched by an agent.

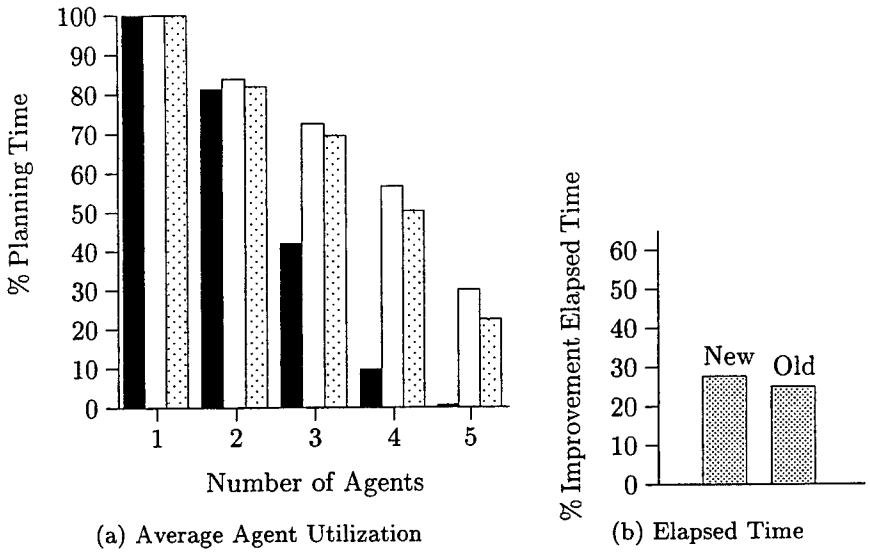
Figure 5.26 shows the results of this set of experiments. [18] The degree to which plan performance can improve depends partially on the particular initial mapping of regions to agents. Some distributions are better than others. To remove the impact of any single particular initial distribution on the results, we ran the same problem using ten random distributions of the regions to planning agents. To get an idea of overall agent utilization across distributions, we averaged our graphs as shown in Figure 5.26 a. For example, this graph shows that on average we were able to utilize three agents 72% of the time with the new heuristic, 69% of the time with the old heuristic, and 42% of the time without a heuristic. To measure the improvement in time to construct the plan, we averaged the percent change in simulated elapsed times as shown in Figure 5.26 b. The results show that employing either heuristic significantly increases agent utilization and improves the overall time to construct a plan when a high degree of agent utilization is possible. Thus, by using our heuristics, agents were able to effectively balance their individual responsibilities for conducting portions of the search. Since the new heuristic resulted in a slightly improved performance over the old heuristic, the new heuristic was used in the rest of our experiments.

5.9.2 Dynamic Self Organization

In addition, we used the same scenario to measure the agents' ability to reorganize the problem distribution using our decision criterion. Since the search spaces are non-overlapping and there is an equal number of planning agents and goals, we would expect to make two observations. First, each agent should eventually take responsibility for one goal, thereby balancing responsibilities. Second, each agent should eventually collect the distinct set of regions that are included in the search space of a plan for its goal, thereby minimizing agent interference.

We ran five trials in which we examined the mapping of regions to agents before and after plan construction. The results are shown in Table 5.1. Each entry for an agent and a

[18] Since our heuristic can only effect the first stage of planning, the %Planning Time and Simulated Elapsed Time refer to the first stage of planning only.



(a) Average Agent Utilization

(b) Elapsed Time

Figure 5.26: Experimental Results - 5 Agents - No Search Space Overlap

distribution has two columns. The left column represents the distribution of regions to that agent before planning, and the right column represents the distribution of regions to that agent after planning. The following abbreviations are used to conserve space: AX = Arm X, RRX = RobotRealm X, TXAY = TableArm TableXArmY, and TRX = TableRealm X. Although each trial began with a different random distribution of region definitions to agents, they all converged to approximately the same distribution through the planning process. Four of the final distributions matched the characteristic distribution we expected to see. The other distribution, number 2, came close with one agent giving up all its regions, and another taking on two goals rather than merely one. Thus, these experiments show that our decision criteria is effective in allowing agents to shift region definitions to appropriate agents.

5.9.3 Effects of Increased Agent Interference

Our next sets of experiments involved varying the overlap of the search spaces for the five goals. Keeping the same number of planning agents and the same number of goals, we looked at four cases: (1) search spaces for the goals with no regions in common - the case above, (2)search spaces for the goals having a few regions in common, (3)search spaces with many regions in common, and (4)search spaces involving exactly the same regions. In these

	Agent									
Dist	A		B		C		D		E	
1	A2 TR1 TR3	A3 RR3 T3A3 TR3	RR2 TR4 T3A3 TR2	A4 RR4 T4A4 TR4	A1 A5 RR1 RR3 T4A4 T5A5	A5 RR5 T5A5 TR5	A4 T2A2 TR5	A2 RR2 T2A2 TR2	A3 RR4 RR5 T1A1	A1 RR1 T1A1 TR1
2	A2 RR5 TR1 TR5	A1 RR1 T1A1 TR1 A2 RR2 T2A2 TR2	A4 RR3 RR4 T4A4 T5A5	A5 RR5 T5A5 TR5	A3 A5 RR1		T1A1 T3A3 TR2 TR3	A3 RR3 T3A3 TR3	A1 RR2 T2A2 TR4	A4 RR4 T4A4 TR4
3	A3 RR4 T5A5	A2 RR2 T2A2 TR2	T4A4	A4 RR4 T4A4 TR4	A1 A5 RR2 RR3 RR5 T2A2 TR5	A5 RR5 T5A5 TR5	A4 TR3	A3 RR3 T3A3 TR3	A2 RR1 T1A1 T3A3 TR1 TR4	A1 RR1 T1A1 TR1
4	RR1 RR3 RR4 RR5 TR3	A5 RR5 T5A5 TR5	A2 T4A4	A4 RR4 T4A4 TR4	T3A3 TR1 TR2	A1 RR1 T1A1 TR1	A4 A5 T2A2 T5A5 TR4	A2 RR2 T2A2 TR2	A1 A3 RR2 T1A1 TR5	A3 RR3 T3A3 TR3
5	A2 RR1 TR1	A1 RR1 T1A1 TR1	A3 A4 RR3	A3 RR3 T3A3 TR3	A1 RR2 RR4 T3A3 T4A4	A4 RR4 T4A4 TR4	A5 T1A1 T5A5 TR3	A5 RR5 T5A5 TR5	RR5 T2A2 TR2 TR4 TR5	A2 RR2 T2A2 TR2

Table 5.1: Results from Experiments on Dynamic Self Organization

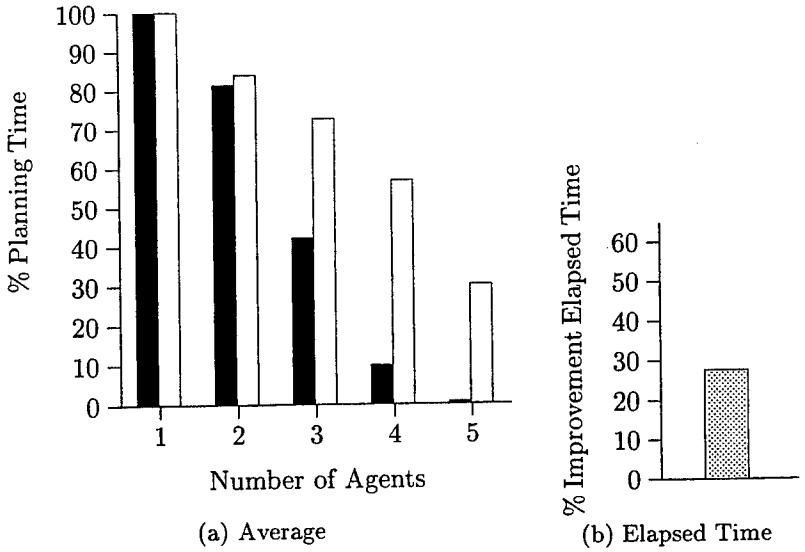


Figure 5.27: Experimental Results - 5 Agents - No Search Space Overlap

four scenarios, there is an increasing potential for one agent to interfere with another by having its search "block" the progress of another agent's search. That is, one search may have to wait for another search to complete in a region before it can continue in that region.

The objective of these experiments was to determine the effectiveness of our heuristic as the inherent agent interference increases. These results are shown in Figures 5.27, 5.28, 5.29, and 5.30. We observe from Figures 5.28 and 5.29 that agents were able to improve their performance in cases with both low and intermediate levels of partial overlap in the search spaces. As one should expect, the degree of improvement in performance is limited by the increased agent interference inherent in the problem. Figure 5.30 demonstrates that for situations in which the search spaces completely overlap, there is a cost associated with determining that reorganization may not be effective. This cost is small, representing an average of 2% increase in the total simulated planning time.

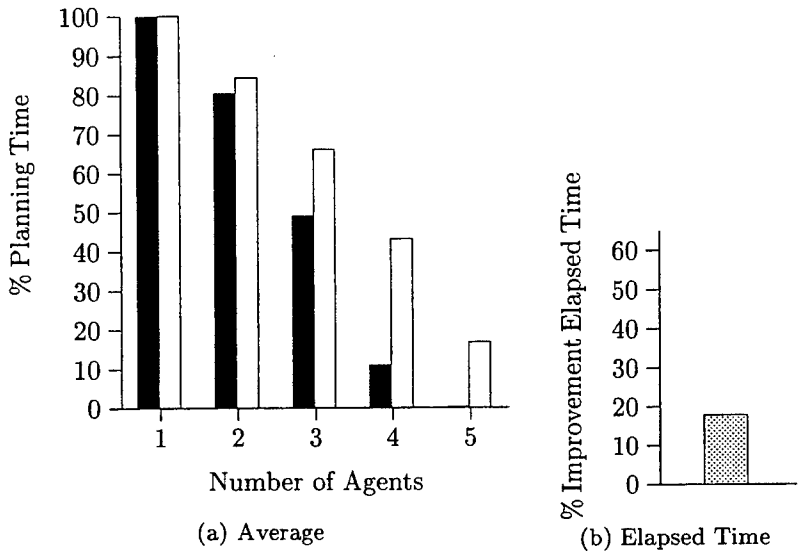


Figure 5.28: Experimental Results - 5 Agents - Low Search Space Overlap

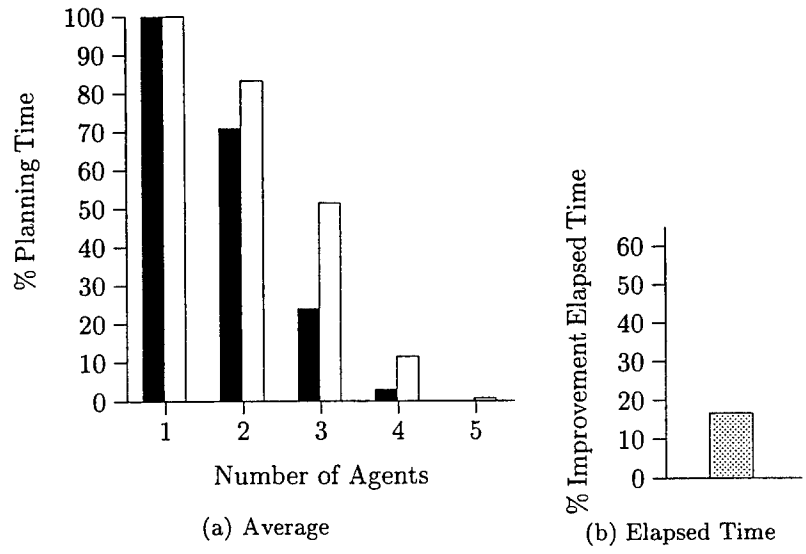


Figure 5.29: Experimental Results - 5 Agents - Intermediate Search Space Overlap

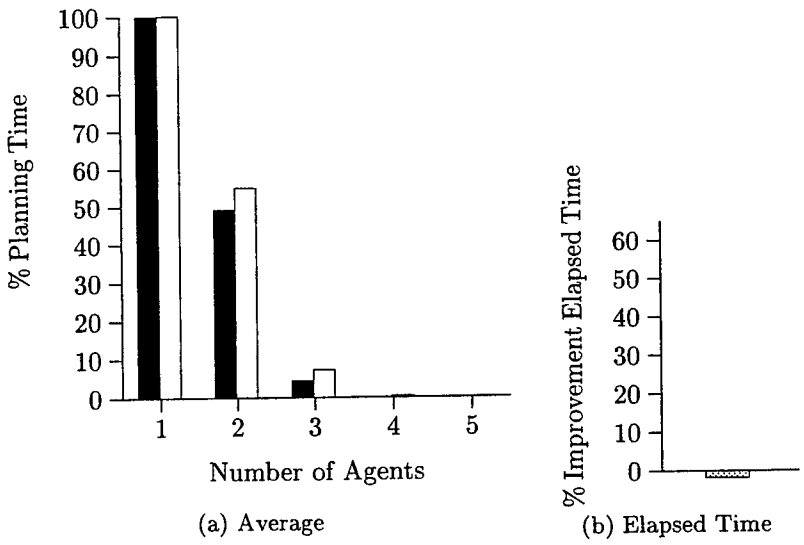


Figure 5.30: Experimental Results - 5 Agents - Total Search Space Overlap

5.9.4 Varying the Goal-vs-Agent Ratio

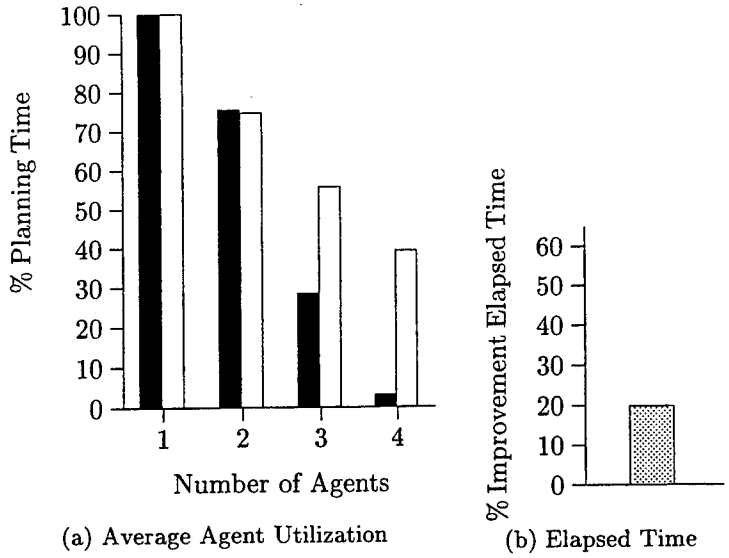
In our final set of experiments, we observed the effect of varying the ratio of the number of goals to the number of planning agents. We repeated the experiments of the previous section for two agents, four agents, seven agents, and ten agents. In each case we ran trials for five random distributions. The results are shown in Figures 5.31 - 5.46. The changes in time to construct a plan and message traffic are are summarized in Figures 5.47 and 5.48.

The results were surprising. As the number of agents grows smaller than the number of goals, as in the two agent case (Figures 5.35-5.38) and the four agent case (Figures 5.31-5.34), we had expected to see a decrease in the ability of our heuristic to improve performance. However, although the overall planning time increased with fewer agents, the improvement we observed in agent utility with the heuristic remained significant. Furthermore, the percent change in simulated elapsed time followed the same pattern as when there were equal numbers of goals and planning agents as shown in Figure 5.47. From this we observe that, although the ratio of goals to agents necessarily restricts the number of searches that can be done in parallel, our heuristic can bring about significantly improved performance within those bounds.

In the cases where the number of planning agents exceeds the number of searches, we also observed significant improvement in agent utility and simulated elapsed time using our heuristic (Figures 5.39-5.46). As in the case with equal numbers of goals and searches, we see a general decrease in agent utility as the inherent agent interference increases. The improvements in agent utility and elapsed simulated time remain significant as with all the other cases. The increased cost in message traffic though, becomes significant.

In Figures 5.47 and 5.48, we compare the percent improvement in planning time and the percent change in the number of messages sent for number of planning agents. The bars are grouped in sets by the number of agents. Each bar within a set represents the percent change with and without the heuristic. There are four bars in each set representing the four cases of search space overlap. They represent the case of no overlap, low overlap, intermediate overlap, and total overlap respectively.

Note that as the number of planning agents increases relative to the number of goals, we observe a significant reduction in the number of messages sent when there is no search space overlap. In these situations, the agents are able to quickly reorganize the regions into their distinct groups for each goal and to reorganize planning responsibilities. Once this is completed, agents no longer need to communicate with one another because each has the



(a) Average Agent Utilization

(b) Elapsed Time

Figure 5.31: Experimental Results - 4 Agents - Non-overlapping Searches

information it needs to complete the plan for the goal for which it has responsibility. Thus the system realizes a reduction in required message traffic because agent interference is reduced. Note however, that this is not the case as the inherent agent interference increases with greater search space overlap. As the inherent agent interference increases, it becomes more and more difficult to find a good organization. This is reflected in increased message traffic. As noted above, this cost becomes quite significant when there are many more agents than goals.

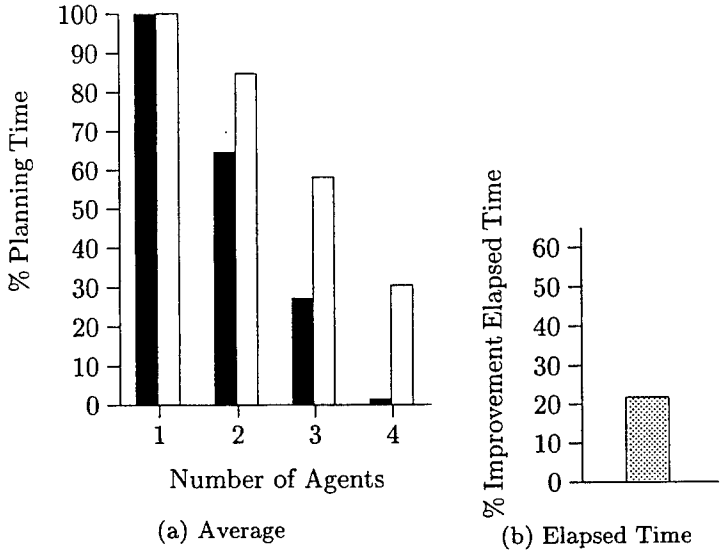


Figure 5.32: Experimental Results - 4 Agents - Low Search Space Overlap

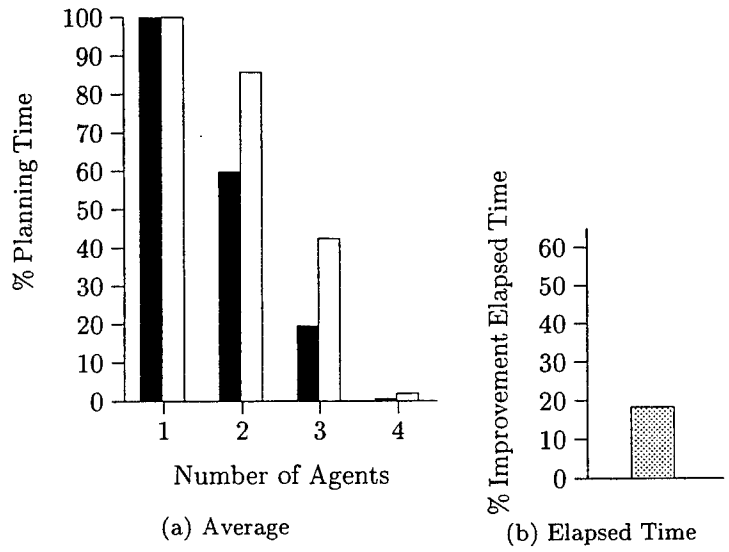


Figure 5.33: Experimental Results - 4 Agents - Intermediate Search Space Overlap

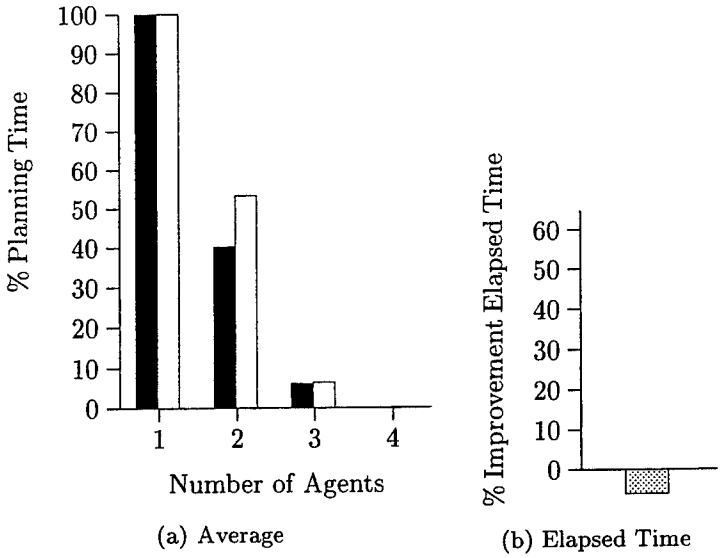


Figure 5.34: Experimental Results - 4 Agent - Total Search Space Overlap

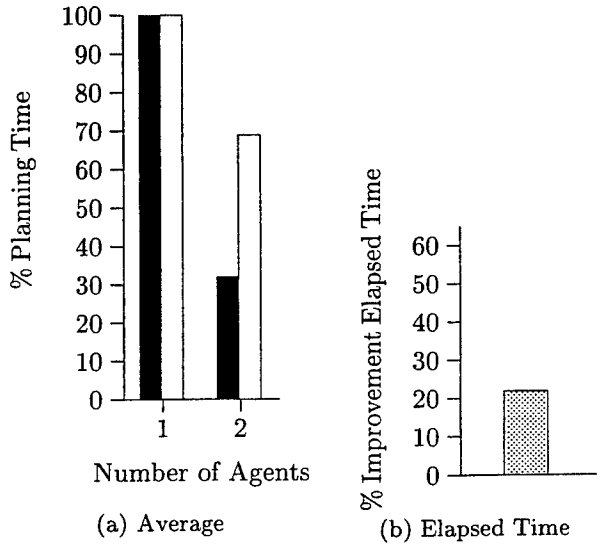


Figure 5.35: Experimental Results - 2 Agent - No Search Space Overlap

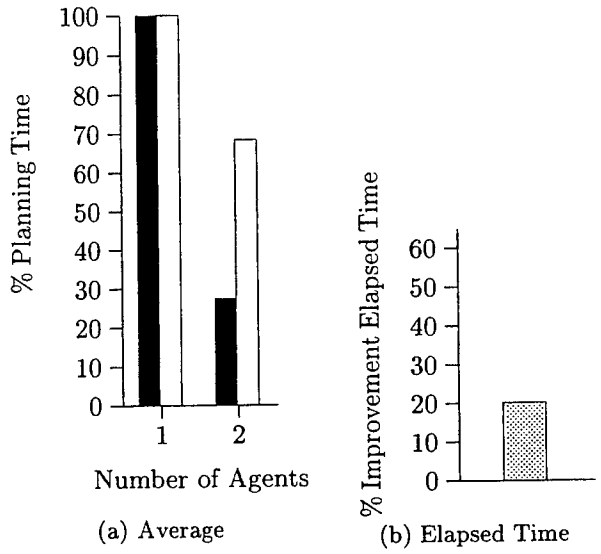


Figure 5.36: Experimental Results - 2 Agent - Low Search Space Overlap

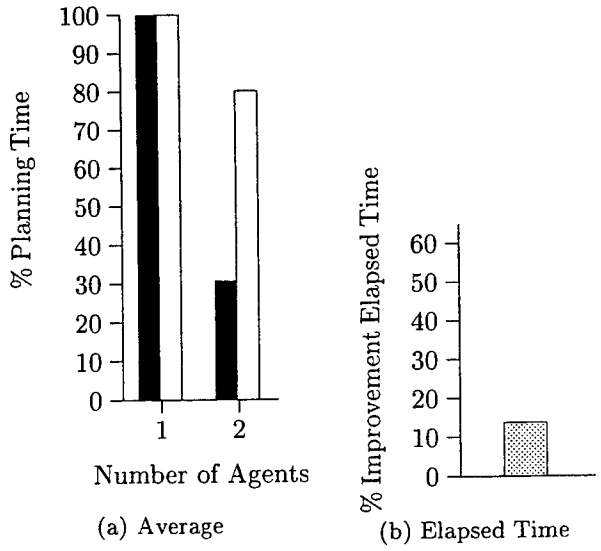


Figure 5.37: Experimental Results - 2 Agent - Intermediate Search Space Overlap

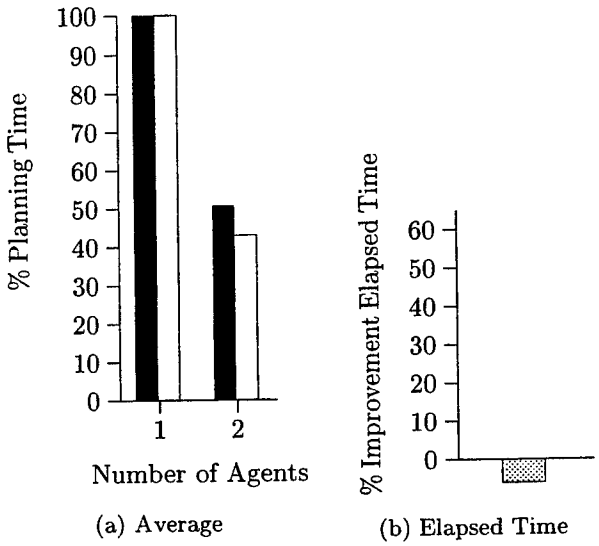


Figure 5.38: Experimental Results - 2 Agent - Total Search Space Overlap

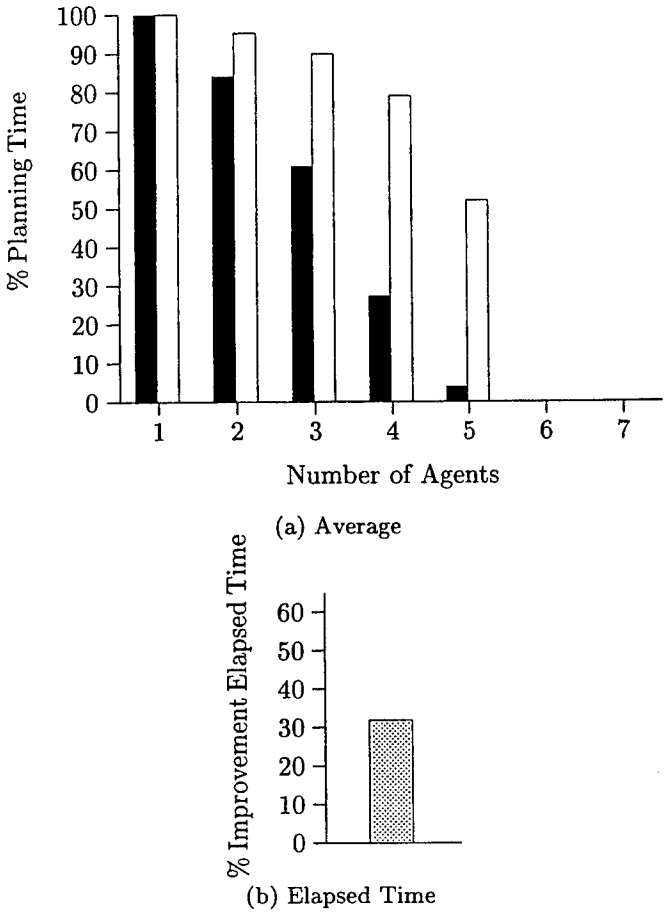


Figure 5.39: Experimental Results - 7 Agent - No Search Space Overlap

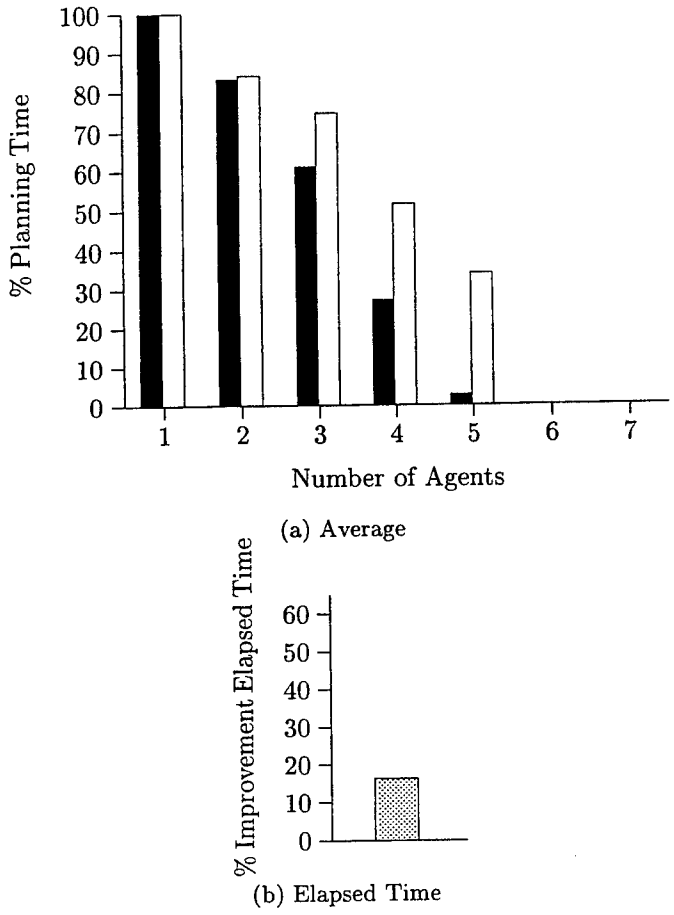


Figure 5.40: Experimental Results - 7 Agent - Low Search Space Overlap

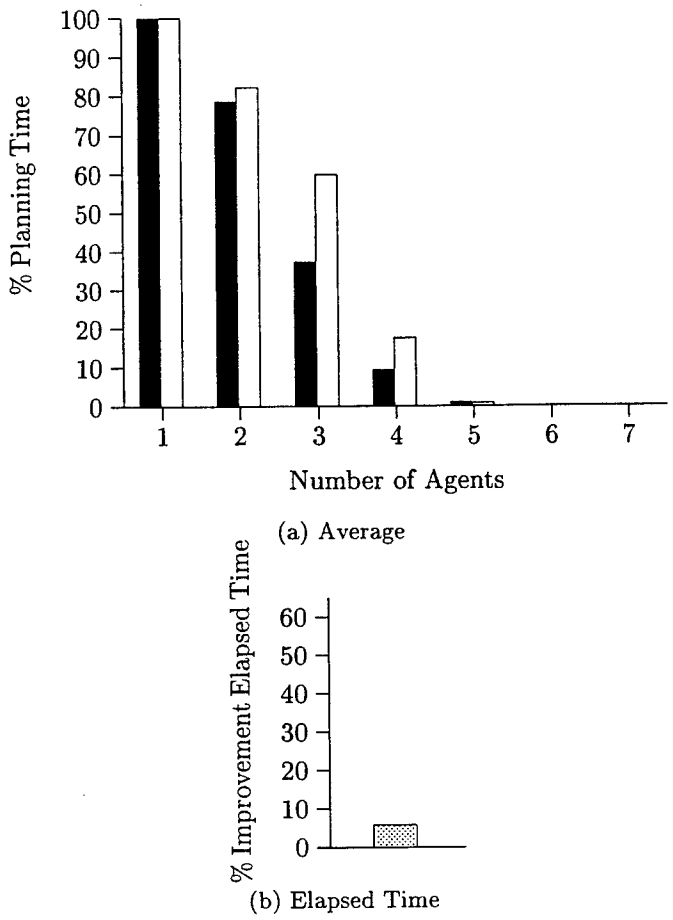


Figure 5.41: Experimental Results - 7 Agent - Intermediate Search Space Overlap

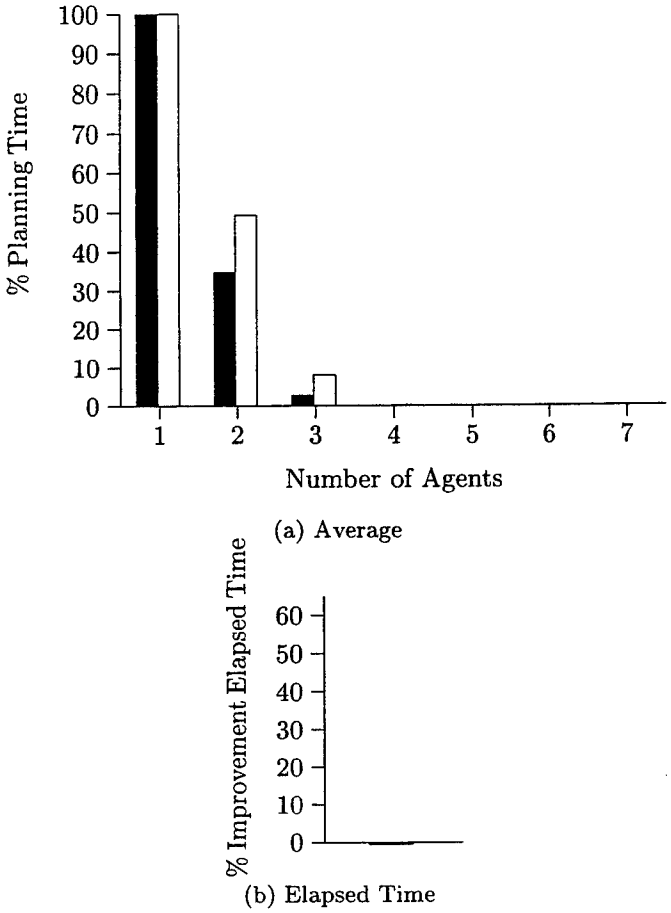


Figure 5.42: Experimental Results - 7 Agent - Total Search Space Overlap

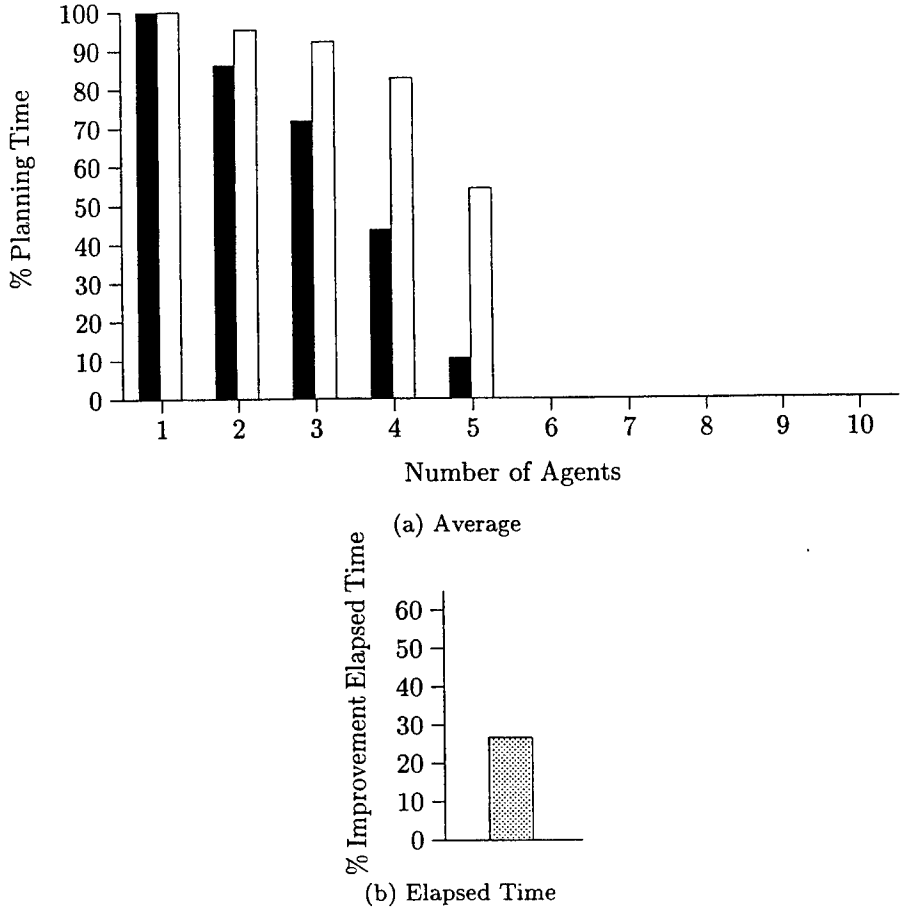


Figure 5.43: Experimental Results - 10 Agent - No Search Space Overlap

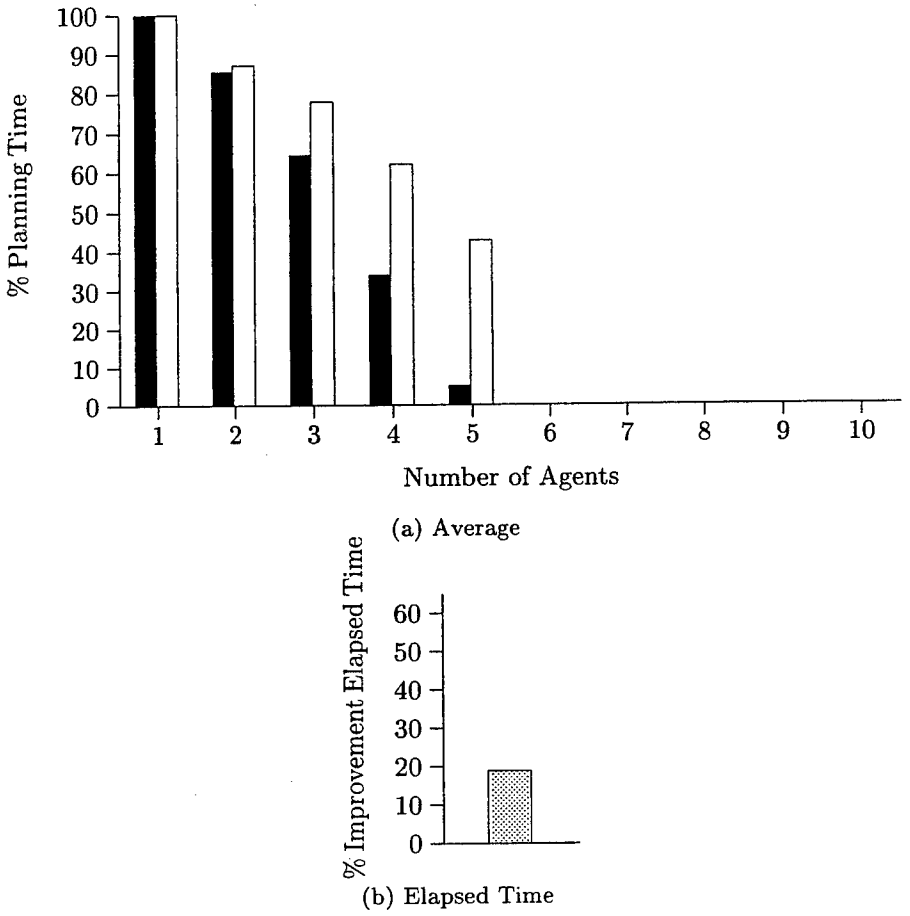


Figure 5.44: Experimental Results - 10 Agent - Low Search Space Overlap

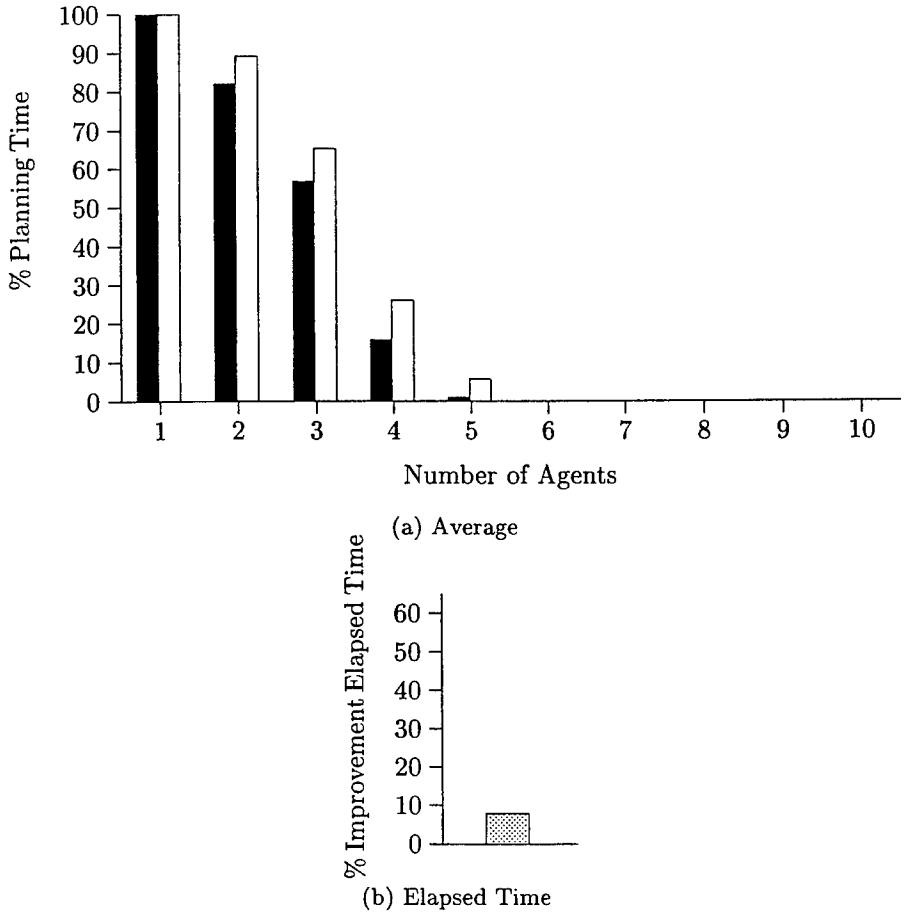


Figure 5.45: Experimental Results - 10 Agent - Intermediate Search Space Overlap

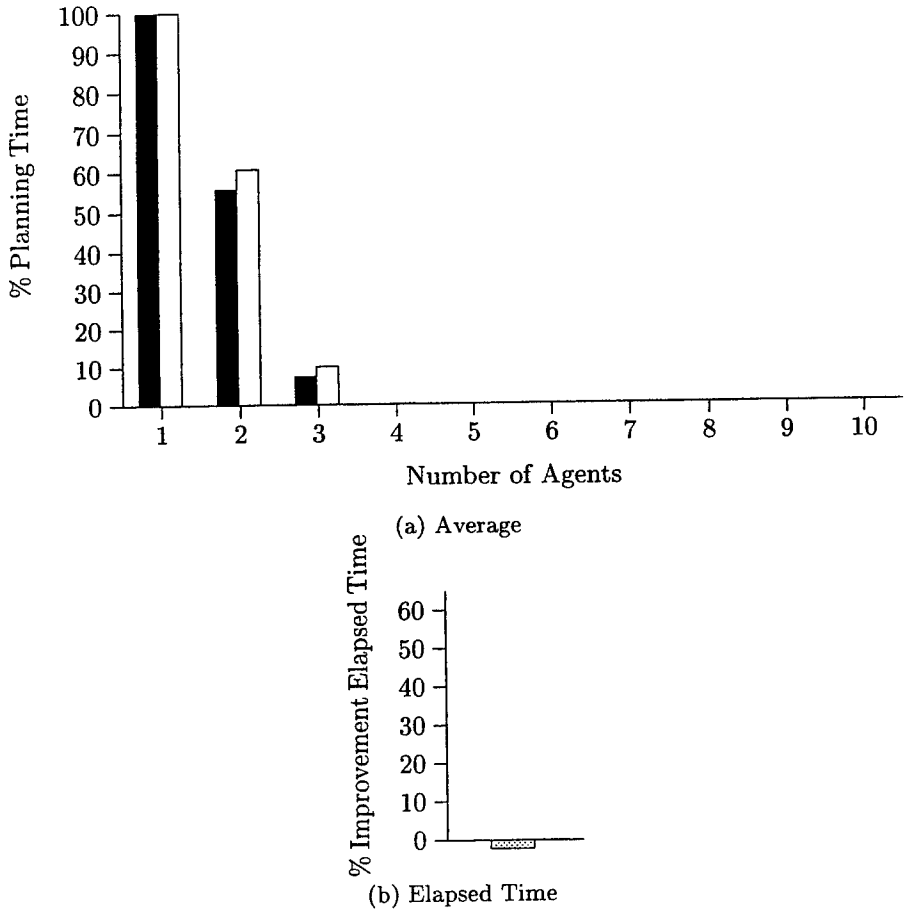


Figure 5.46: Experimental Results - 10 Agent - Total Search Space Overlap

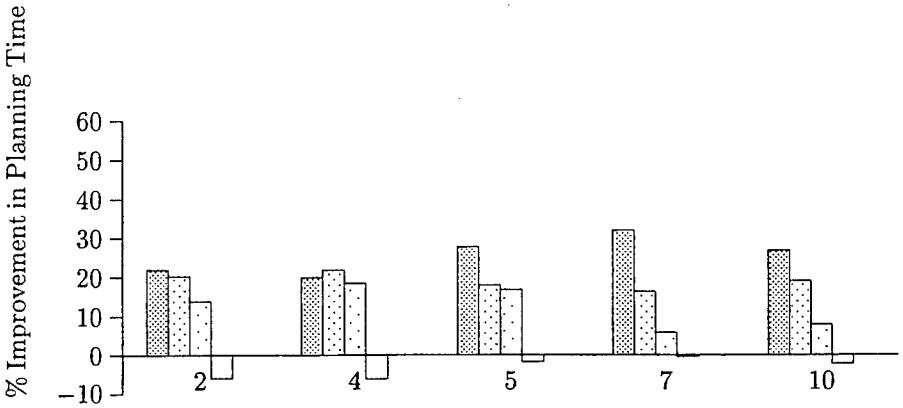


Figure 5.47: Comparison of Planning Time Improvement

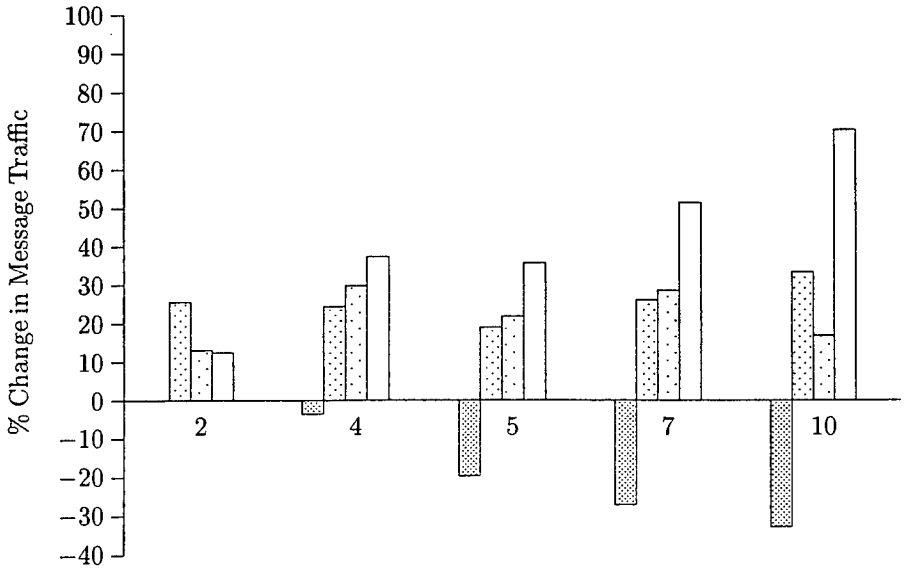


Figure 5.48: Comparison of Change in Message Traffic Costs

Chapter 6

Link Activation Scheduling

6.1 Introduction

For communication networks which utilize some form of multiaccess channel, a means for controlling channel access must be provided. In this chapter we discuss multihop shared channel networks having a prescribed connectivity matrix. An example of such a network is a multihop packet radio network in which each activation of a link between two nodes must be scheduled so as not to produce conflicts with activations on other links.

Link scheduling can be viewed as a constraint satisfaction problem with constraints used to prevent link activation conflicts and to ensure that each link is scheduled for a prescribed number of activations. In previous work [67, 66], we have developed an approach to solving constraint satisfaction problems in distributed environments. In this chapter we present a formulation of link scheduling as a distributed constraint satisfaction problem and explore the impact of varying the organization of network control on link activation scheduling.

The link scheduling problem is known to be NP-complete under most conditions [1, 36], but variations of the problem have been introduced [36, 9] which can be solved in polynomial time. Even in these cases the computational cost for scheduling of large networks is substantial. Furthermore, in networks with dynamic connectivity, a centralized decision making node may not have correct global data needed for link activation scheduling. To mitigate these problems, our approach is to divide the larger, network-wide scheduling

problem into smaller, independent scheduling problems. These independent scheduling problems are solved in parallel by a set of network control agents acting autonomously. Links shared between two agents are then scheduled by a process of agent-to-agent coordination in order to produce a complete schedule. Although the resulting schedules are not optimal, they satisfy the property of *maximal slot assignment* [9] which guarantees that for each time slot no additional node could schedule a link activation without conflict. Thus these schedules have high channel utilization.

In the next section, we define the problem and compare our work to related results from the literature. Section 3 describes DCONSA – a Distributed CONstraint-based planner for Semi-autonomous Agents – and shows how link scheduling may be formulated as a constraint satisfaction problem in the context of DCONSA. Section 4 presents the experimental results which illustrate the tradeoffs to be made in solving the link activation problem as network control organization is varied from centralized to fully distributed. These results clearly demonstrate a benefit to be gained using a distributed approach which is intermediate between the extremes of a single agent centralized model and the one agent for every node, fully distributed model.

6.2 The Scheduling Problem

We define the problem of scheduling link activations as follows. Given a graph $G = (V, E)$ expressing network topology, a traffic demand set TD_{SD} of source-destination pairs, and a multihop path of link activations to satisfy each TD_{ij}, produce a link activation schedule that is conflict free. This is the same problem formulation as in Barnhart et al. [3], although our approach to solving the problem is not related to their neural net solution. Following their problem definition, we also examine in this chapter the nonsequential activation scheduling (NAS) problem. In this version of the problem the goal is to determine a schedule of time slots in which each link is allotted a number of time slots which corresponds to the number of activations required to satisfy the traffic demand set.

In reviewing the literature, one can find the link scheduling problem posed in several different ways, each with a different set of assumptions. We are interested in the distributed forms of this problem and will focus our discussion on these implementations. In [68], Post et al. present a distributed version of their heuristic algorithm. It involves the distribution of all network topology and link transmission demands to each node in the system. Each node can then perform the centralized version of the algorithm for itself. Kershenbaum and Post [43] present an algorithm that requires minimal information but does not use traffic

demands and produces nonoptimal schedules. In [23] and [2] distributed algorithms for scheduling are proposed but these do not take into account traffic demands. Cidon and Sidi [9] present an algorithm that finds a maximal assignment of link activations for a slot, and then explore ways to extend this algorithm over a sequence of time slots.

In general these distributed algorithms operate under the assumption that the problem is fully distributed at a nodal scale - i.e. link activations are assigned to time slots by each transmitting node. As pointed out in [2], these fully distributed algorithms must tradeoff considerable communication overhead against the quality of the schedules produced. In order to achieve optimal or near optimal schedules each node must have global or nearly global information. Thus, there is an interest in studying alternative network control organizations which allow for a degree of distributed decision making using only limited connectivity information. Our approach provides a uniform examination of various network control organizations as applied to the link allocation scheduling problem.

Many of the algorithms are closely tied to the assumption about which types of conflicts are allowed and can not adapt to changes in this assumption. We use a framework which can facilitate changing the allowed forms of conflict without changing the scheduling algorithm itself.

Finally, in comparing the quality of schedules obtained in our results with those cited above, we are able to guarantee maximal slot assignments but not maximum slot assignments as defined in [9].

6.3 Link Scheduling as a Problem for DCONSA

In this section, we present our formulation of the link scheduling problem in the DCONSA planning system [67, 66]. DCONSA is a distributed planning system that employs the GEM model [46] for problem representation. The flexibility of the DCONSA system allows us to investigate various network organizations in a single framework without changing the underlying scheduling process. We first give an overview of DCONSA, and then we describe how the link scheduling problem is represented in DCONSA.

In DCONSA, a plan is represented as a set of events and relations on those events. Planning is viewed as a constraint satisfaction problem in which constraints are used to describe goals and required relations on events. Plan construction involves the incremental

addition of new events and relations to a plan until it conforms to a given set of constraints. Initially, the plan may be empty or it may consist of a set of initial events and relations.

DCONSA is designed to develop plans in a distributed environment. This environment consists of semi-autonomous agents, each having local knowledge and control of local resources. These agents must be able to coordinate individual choices and actions so as to cooperate in achieving global goals. We achieve this coordination by organizing the domain knowledge in a formal structure which is distributed across a set of planning agents.

Domain knowledge, formulated as a constraint satisfaction problem, is organized by the concept of a region. A region is a conceptual means to encapsulate a set of events with a set of constraints that relate those events. A region may be either a simple *element* or a *group*. An element is a primitive unit of organization. Every event belongs to a unique element. The set of constraints which pertain only to the events of a single element are included as part of the element description. To reason about constraints which relate events in distinct elements, we need to incorporate a higher level of organization - a group. A group is a set of regions (elements and/or groups) and a set of constraints which relate events in its member regions. Thus, a planning problem can be structured into a set of regional constraint satisfaction problems. A plan is created by conducting a set of bounded region searches. Region searches are bounded, because only the events and constraints within that region are considered during the search. An acceptable plan is one that satisfies all constraints in all regions.

We can use this model to represent the link scheduling problem as follows. We model link activations as events, and we define a relation, $Assign(ev, ts)$, that associates a link activation with a time slot. This relation represents the scheduling of the link activation event during that time slot. A plan is a set of *Assign* relations on a set of link activation events. A plan is complete if each link activation event has an associated time slot. Network links are modeled as elements whose member events are its activations. Each network node is modeled as a group comprised of link elements. To distribute the problem across a set of planning agents, we divide the network into areas. Each area is modeled formally as a group comprised of node groups. Thus, each network area represents a subset of the network whose activations will be scheduled by a particular DCONSA planning agent. Network links which physically join nodes in distinct areas are called shared links. A shared link belongs to two node groups which in turn are members of distinct network areas. All other links are called interior links. Activation events on shared links represent planning activity which will necessarily have to be coordinated between the planning agents associated with their respective network areas. However, each agents' view of this link is limited - nothing is known about its connectivity in the other area.

As described earlier, our basic approach is to break the larger scheduling problem into smaller independent scheduling problems, plan these schedules in parallel, and then to serially schedule the activations which interconnect the independent schedules. Using the organization described above, we can identify the independent scheduling problems. Each agent can independently schedule the activations of the interior links of its network area, since time slot assignments for these links do not have to be coordinated with any other agent. However, as noted earlier, the scheduling of link activations on shared links does require agent coordination and thus, these will be scheduled serially. Furthermore, since two agents that share a link have limited views of that link, each will have to coordinate proposed time slot assignments of its activations with their respective interior schedules. Therefore, to complete the representation of problem structure, we also define a boundary group to represent the interface between problem solving in different network areas. The members of a boundary group are the node groups of a network area which have link elements that are shared with other network areas. Each planning agent is associated with a network area group and a boundary group.

To complete the model, we need to furnish the constraints which describe an acceptable schedule, and which will model the problem solving activity as described above. We accomplish this with the area group constraints shown in Table 6.1 and the boundary group constraints shown in Table 6.2.

1. All activations on *interior* links are scheduled.
2. No two activation events which are members of the same link can be scheduled for the same time slot.
3. No two activations events which are members of the same node can be scheduled for the same time slot.

Table 6.1: Area Group Constraints

1. All activations on *shared* links are scheduled.
2. No two activation events which are members of the same link can be scheduled for the same time slot.
3. No two activations events which are members of the same node can be scheduled for the same time slot.

Table 6.2: Boundary Group Constraints

Using the area constraints, each DCONSA planning agent can schedule the interior activations of its associated area group. Area Constraint #1 ensures that links which are shared among areas are not scheduled. Thus, the scheduling of interior activations in distinct network areas is guaranteed to be independent and can be executed in parallel. Once the planning agents detect that they have completed their interior schedules, they can begin to serially schedule the activations on shared links. This is accomplished using the boundary group constraints shown in Table 6.2.

Since an agent's view of its shared links is limited, a time slot it selects for a given activation on a shared link may not be compatible with the interior schedule of another agent. If this situation arises, the regional search in the second agent's boundary region will fail, and search will backtrack to the first agent which scheduled the activation. The first agent will then select another time slot. This process continues until a slot assignment which is compatible in both schedules is found, or if none exists, a new time slot is appended to the schedule. This guarantees that slot assignments made during the parallel search need not be retracted.

An advantage of this model is the flexibility provided by expressing conflict types as constraints. The constraints we have presented in this section prevent primary conflicts from occurring in the schedule - see Area Constraints #2 and #3 and Boundary Constraints #2 and #3. However, the set of conflicts which are not allowed can easily be expanded to include other forms of conflict (such as secondary conflicts [3, 9]) by modifying the set of constraints in the area and boundary groups. This addition does not require any changes to the overall scheduling algorithm. Although not pursued in this report, this permits us to explore the effects of various conflict types on the schedules produced.

Area and boundary groups allow us to organize a given network in various ways. We can model a centralized scheduling problem by representing the entire network in a single area with no boundary group. We can model a completely distributed scheduling problem by representing each node as an area group and a boundary group. In this case, there is no parallel search because all links are shared. Link activations are scheduled through message passing among agents during planning in the boundary groups. We can also model various distributions between these two extremes by varying the size of the network areas and their mapping to physical network topology. In the next section, we describe the experiments we have performed using these various organizations.

6.4 Experiments

Using the model described in Section 6.3, we have performed experiments to investigate distributed link activation scheduling using various network organizations. In particular we are interested in the tradeoffs between increased parallelization of the problem with the decrease in knowledge to make time slot assignments. As the network is divided into a greater number of areas, more of the problem can be done in parallel thus decreasing the amount of time to create the schedule. However, two factors work against the improved speed of this parallelization. Depending upon the sparseness of the network, an increase in the number of network areas (as we have defined them) potentially results in an increase of shared link activations. Thus, more of the problem is pushed from the parallel search to the serial search. Furthermore, time slot assignments in the serial search are more likely to fail because they are based upon limited information. Therefore, not only do we increase the time to complete a plan because of a decrease in parallelization, we also potentially increase the time due to increased backtracking during search.

To investigate these tradeoffs, we have implemented the ideas presented in this chapter using a scheduling example from [3] shown in Figure 6.1. Each arrow represents a link activation between two nodes. Thus, the two arrows between nodes 4 and 5 indicate that the link connecting these nodes has two activations to be scheduled. We create a schedule for this network using 1, 2, 4, 6, and 23 planning agents. In the 1 agent case, the entire network is modeled as a single area group. Thus, slot assignments are made with complete knowledge. In the cases with 2, 4, and 6 agents, the network is divided into the respective number of network areas as shown in Figures 6.2, 6.3, and 6.4. The 23 agent case represents this division into areas taken to its extreme with each area consisting of a single node. Thus, no interior scheduling occurs and all slot assignments are made with limited information. Table 6.3 presents the simulated time to create the schedule, the length of the schedule created in time slots, and relevant characteristics of the network organization. Schedule creation times have been normalized to the simulation time of the single agent, centralized organization.

As expected, the tradeoffs discussed previously are present in the experimental results. The time to create a schedule improves from the single agent case to the four agent case, but decreases in the last two cases. Note the ratio of the average number of interior activations per region to the total number of shared activations. This is a rough indication of how the scheduling process is shifting from the parallel search to the serial search. In the two agent case the ratio is about 2:1 parallel to serial, then about 1:2 in the four agent case, and nearly 1:5 in the six agent search. As discussed earlier, the method we employ guarantees a schedule with time slots that are maximal but not maximum as described in [9]. In

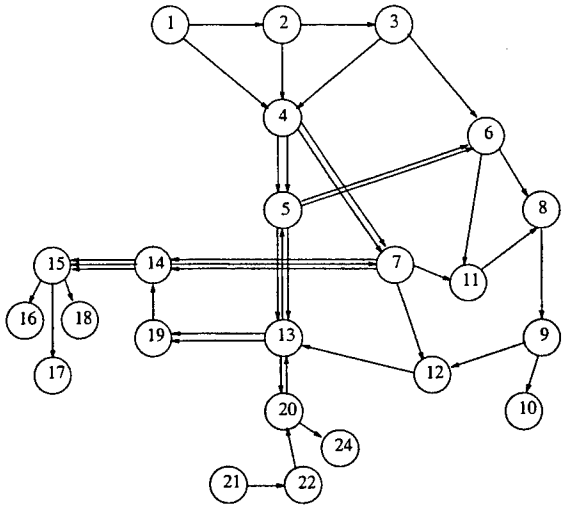


Figure 6.1: Example Network

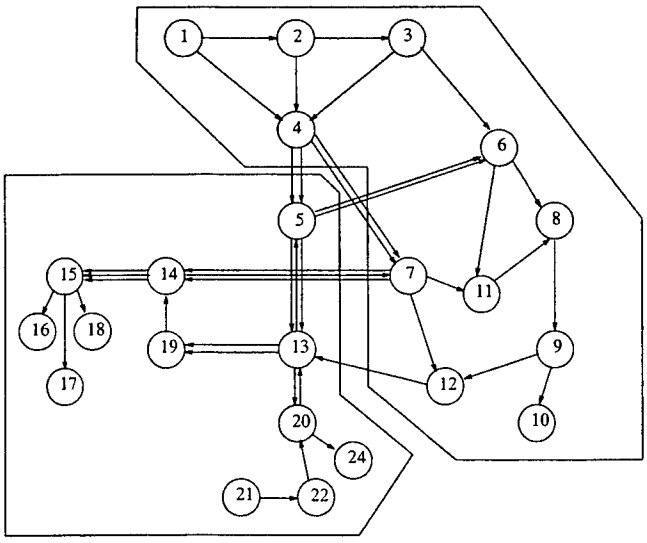


Figure 6.2: Network Organization: Two Agents

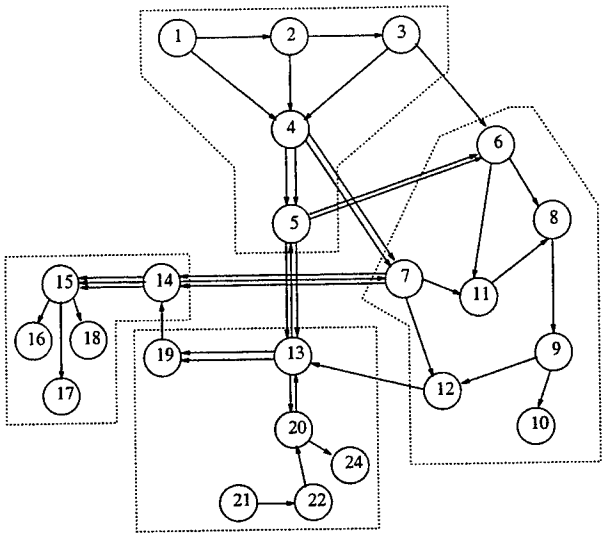


Figure 6.3: Network Organization: Four Agents

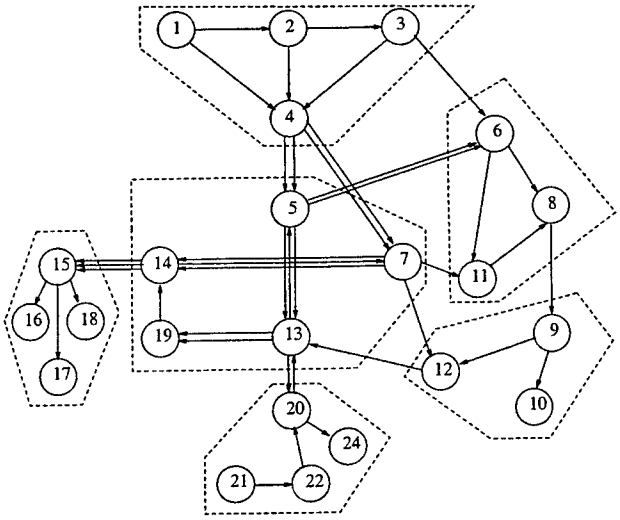


Figure 6.4: Network Organization: Six Agents

Agents	Simulation Time	Schedule Length	Avg Int Activations per Region	Shared Link Activations
1	1.00	8	41	0
2	.57	9	16.5	8
4	.53	9	7	13
6	.74	8	3.83	18
23	1.47	10	0	41

Table 6.3: Scheduling Results and Organization Characteristics

this example the optimum schedule length is eight time slots and our method has found schedules at or close to that length.

6.5 Results

In this chapter, we have explored the effects of distributing the link scheduling problem using varying network organizations. By organizing the network into areas with independent scheduling problems, we can parallelize portions of the problem and then schedule the link activations which are not included in these independent problems. Using the distributed planner DCONSA, we have conducted experiments varying the size of these network areas. These empirical results illustrate that the overall time to create a schedule can be decreased by this parallelization. However, there is a point at which the advantage of the parallelization is overshadowed by increased interconnectivity among the independent areas and limited knowledge resulting in potentially poor time slot assignments. This point depends in general on the ratio of the link activations interior to an area and the shared link activations. Another strong determiner of this tradeoff point which we have not explored here is a measure of "fit" between the network topology and the organizational structure.

Chapter 7

Conclusions

The overall results have been well documented in summary form in section 1.3 and in detail in each of the sections discussing one aspect of this research. In this chapter we discuss some of the limitations and unresolved problem areas of this research, and we identify significant issues for future work.

In the area of testbed development, we believe our proposed design of DiST was sound. Instead of attempting to reuse existing software, it should be designed in detail and implemented from scratch. The decision to attempt to use existing software from a third party was the single worst decision we made.

The Distributed Problem Solving systems discussed in the third chapter have both made use of the DARES automated reasoning system in a straightforward and conventional way. That is, each distributed system assumed a one-to-one correspondence between its agents and the reasoning agents of the DARES system. Furthermore, only distributed reasoning tasks which involved a single theorem were considered. The structure of DARES is flexible enough however, to admit a number of other possible problem solving paradigms. For example, suppose a distributed problem has arisen in which it is known that every solution must follow a generally predictable sequence of steps. Each step toward a solution can be thought of as a solution sub-goal and given to a separate DARES reasoning agent as a theorem to prove. The various theorems representing a solution and its sub-goals, can then be "linked" together to form a *problem-solving cascade* by supplying each reasoning agent with the same theorem tag. Conceptually, this technique amounts to imposing a hierarchical organization on the search space in which each "successive" reasoning agent can incorporate

the intermediate results obtained by the "lower-level" reasoning agents. Further research into the feasibility and usefulness of this problem solving organization, as well as others, may lead to drastic reductions in the "effective" size of a search space. Proven reductions of this kind, would make the use of distributed automated reasoning all that much more appealing.

In order to make effective use of an automated reasoning system, it is necessary to ensure an absolute consistency of the encoded knowledge. Without a consistent base of knowledge, there is no way of readily determining whether or not the results obtained from a reasoning task may be considered valid. For applications of automated reasoning in domains which are both dynamic *and* distributed, the problem is significantly compounded. Clearly, some form of distributed "truth maintenance" is necessary. For a highly dynamic distributed system, such as a communication network, it may be more practical for a truth maintenance system to support a grade of knowledge believability rather than strictly enforcing a two-value system. Presumably such a system would be based on some form of Bayesian probability. When a distributed agent initiates a request for knowledge it requires to formulate axioms for a reasoning task, it registers its use of this knowledge with the truth maintenance system. An agent could handle the graded believability of this knowledge in two ways; 1) use a limiting function to reduce the belief values of the obtained knowledge to a binary system, or 2) incorporate the believability of each piece of knowledge into the axioms it will generate. Certainly, axioms generated by the first method would be the easiest for an automated reasoning system to handle, but it may not be possible to assure consistent axiomatic knowledge. At the other extreme, selecting the second choice would require that a reasoning system be able to cope with the inherent uncertainty of the supplied "axioms."

In either case, when the truth maintenance system has detected that knowledge registered by a distributed agent has changed, it notifies the agent of the knowledge which is currently out of date. The agent must then assess the situation, and if necessary, formulate an updated set of axioms and notify its respective reasoning agent. Mechanisms need to be devised which would ease (if not automate) the knowledge updating procedures between the truth maintenance system and distributed agent, and likewise between the distributed agent and the automated reasoning agent. For distributed agents built around an expert system shell, like TESS, it may be possible to incorporate a *direct* truth maintenance interface into the shell's control structure. In this way, a change in knowledge status would immediately cause a re-formulation of the necessary axioms which would then be sent directly to the automated reasoning agent. A modification of the TREAT algorithm may make such an interface possible while still retaining production system efficiency.

The work of analyzing how hyper-resolution affects the DARES architecture is not

complete. One item we have not explored is the forward progress heuristics behavior and how it affects the reasoning process. In most of our clause sets the importation of knowledge was the result of an inability to produce new resolvents, not the forward progress heuristic. The forward progress heuristic is important for SHYRLI to be "complete" (given a problem where there is a contradiction, complete means that the contradiction is guaranteed to be found). An analysis of this heuristic and others would be interesting.

The effect that the amount of private predicates in a network of agents has on the performance of the reasoning system has not been explored. It is our intuitive guess that a large number of private predicates would increase the performance of the reasoning system. Private predicates imply that there is a portion of the problem that can be resolved locally without communicating with other agents. This fits in with the idea of SHYRLI agents having a functional distribution. This would lead to longer chains of local reasoning, thus increasing performance.

Voluntary exportation of knowledge is another interesting concept that could be explored. Coupled with the idea of public and private predicates this concept could be easily implemented. A small set of global predicates (i.e. information that may be shared with other agents) and a large set of private predicates would make the choice of voluntarily exporting knowledge easy. An agent would know to export knowledge if a hyper-resolvent contained only global predicates. If the number of global predicates is large, however, knowing when to export knowledge voluntarily becomes difficult.

SHYRLI provides a baseline case of a distributed reasoning system that incorporates no domain knowledge. SHYRLI provides a testbed that can be used to test heuristics that use domain specific information to help select what to communicate and which lines of reasoning to follow. The baseline case could be used as a point of comparison between different strategies for managing coordination and interaction in order to increase system performance.

The flexibility of DCONSA's distributed architecture permits the modeling of inherently distributed systems with fixed organizations. Although not focused upon here, such modeling can be useful to point out advantages and disagnose disadvantages with an existing system. DCONSA could also be used to evaluate different organizations of a problem as was done in the link activation scheduling problem. If problem characteristics change as new problems are given to a set of agents, a single organization will most likely not be adequate to improve performance. By incorporating organizational self design into distributed planning, DCONSA can also model distributed autonomous agents that dynamically reorganize to improve performance *during* planning. Thus agents could discover on their own

the organization that best enhances performance for the current problem.

Bibliography

[1] E. Arikan. Some complexity results about packet radio networks. *IEEE Transactions on Information Theory*, IT-30:681–685, 1984.

[2] D. J. Baker, A. Ephremides, and J. A. Flynn. The design and simulation of a mobile radio network with distributed control. *IEEE Journal on Selected Areas in Communications*, SAC-2(1):226–237, January 1984.

[3] C. Barnhart, J. Wieselthier, and A. Ephremides. Neural network techniques for scheduling and routing `problems` in multihop radio networks. In *Proceedings of the IEEE Military Communications Conference (MILCOM91)*, pages 0407–0413, 1991.

[4] R. S. Boyer and J. S. Moore. *A Computational Logic.* Academic Press, New York., 1979.

[5] Lee Brownston, Robert Farrell, Elaine Kant, and Nancy Martin. *Programming Expert Systems in OPS5: An Introduction to Rule-Based Programming.* Artificial Intelligence Series. Addison-Wesley Publishing Company, Inc., Reading, Massachusetts, 1985.

[6] S. Cammarata, D. McArthur, and R. Steeb. Strategies of cooperation in distributed problem solving. In *Proceedings of the Eighth International Joint Conference on Artificial Intelligence (IJCAI83)*, pages 767–770, 1983.

[7] B. Chandrasekaran. Natural and social system metaphors for distributed problem solving: Introduction to the issue. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-11(1):1–5, January 1981.

[8] C. L. Chang and R. C. Lee. *Symbolic Logic and Mechanical Theorem Proving.* Academic Press, New York, 1973.

[9] I. Cidon and M. Sidi. Distributed assignment algorithms for multihop packet radio networks. *IEEE Transactions on Computers*, 38(10):1353–1361, October 1989.

[10] M. P. Cline. *A Fast Parallel Algorithm for N-ary Unification with A.I. Applications.* PhD thesis, Clarkson University, Potsdam, NY 13676, April 1989.

[11] S. E. Conry, D. J. MacIntosh, and R. A. Meyer. DARES: A Distributed Automated REasoning System. In *Proceedings of the National Conference on Artificial Intelligence (AAAI-90)*, pages 78–85, August 1990.

[12] S. E. Conry, R. A. Meyer, and V. R. Lesser. Multistage negotiation in distributed planning. In *Readings in Distributed Artificial Intelligence.* Morgan Kaufman Publishers, California, August 1988.

[13] Susan E. Conry, Robert A. Meyer, and Victor R. Lesser. Multistage negotiation in distributed planning. In A. Bond and L. Gasser, editors, *Readings in Distributed Artificial Intelligence.* Morgan Kaufman Publishers, California, August 1988.

[14] Susan E. Conry, Robert A. Meyer, and Randall P. Pope. Mechanisms for assessing nonlocal impact of local decisions in distributed planning. In M. Huhns and L. Gasser, editors, *Distributed Artificial Intelligence, Volume II.* Pittman Publishing, Ltd and Morgan Kaufman Publishers, August 1989.

[15] Susan E. Conry, Robert A. Meyer, and Janice E. Searleman. A shared knowledge base for independent problem solving agents. In *IEEE Proceedings of the Expert Systems in Government Symposium*, pages 178–186, Washington, D.C., 1985. IEEE Computer Society Press. McLean, Virginia.

[16] Daniel D. Corkill. Hierarchical planning in a distributed environment. In *Proceedings of the Sixth International Joint Conference on Artificial Intelligence (IJCAI79)*, pages 168–175, 1979.

[17] M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the Association for Computing Machinery*, 7(3):201–215, March 1960.

[18] Randall Davis and Reid G. Smith. Negotiation as a metaphor for distributed problem solving. *Artificial Intelligence*, 20:63–109, January 1983.

[19] Keith S. Decker, Edmund H. Durfee, and Victor R. Lesser. Evaluating research in cooperative distributed problem solving. In Les Gasser and Michael N. Huhns, editors, *Distributed Artificial Intelligence Volume II*, chapter 19, pages 485–519. Pitman Publishing and Morgan Kaufman Publishers, Inc., 1989.

[20] E. H. Durfee, V. R. Lesser, and D. D. Corkill. Cooperation through communication in a distributed problem solving network. In M. Huhns, editor, *Distributed Artificial Intelligence.* Pittman Publishing, Ltd and Morgan Kaufman Publishers, 1987.

[21] E. H. Durfee and T. A. Montgomery. Mice: A flexible testbed for intelligent coordination experiments. In *9th Annual Distributed Artificial Intelligence Workshop.* American Association for Artificial Intelligence (AAAI), September 1989.

[22] Edmund H. Durfee and Victor R. Lesser. Using partial global plans to coordinate distributed problem solvers. In *Proceedings of the Tenth International Joint Conference on Artificial Intelligence (IJCAI87)*, pages 875–883, 1987.

[23] A. Ephremides, J. Wieselthier, and D. Baker. A design concept for reliable mobile radio networks with frequency hopping signaling. *Proceedings of the IEEE*, 75(1), January 1987.

[24] G. W. Ernst and A. Newell. *GPS: A Case Study in Generality and Problem Solving.* Academic Press, New York, NY, 1969.

[25] R. E. Fikes and N. J. Nilsson. Strips: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2(3-4):189–208, Winter 1971.

[26] Charles L. Forgy. RETE: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*, 19:17–37, 1982.

[27] Mark S. Fox. An organizational view of distributed systems. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-11(1):70–80, January 1981.

[28] L. Gasser, C. Braganza, and N. Herman. MACE: A flexible testbed for distributed AI research. In M. Huhns, editor, *Distributed Artificial Intelligence.* Pittman Publishing, Ltd and Morgan Kaufman Publishers, 1987.

[29] L. Gasser and M. Huhns. Themes in distributed artificial intelligence research. In M. Huhns and L. Gasser, editors, *Distributed Artificial Intelligence, Volume II.* Pittman Publishing, Ltd and Morgan Kaufman Publishers, August 1989.

[30] Michail R. Genesereth and Nils J. Nilsson. *Logical Foundations of Artificial Intelligence.* Morgan Kaufmann Publishers, Inc., 95 First Street Los Altos, California 94022, 1987.

[31] Michael Georgeff. Communication and interaction in multi-agent planning. In *Proceedings of the Third National Conference on Artificial Intelligence (AAAI83)*, pages 125–129, August 1983.

[32] Michael Georgeff. A theory of action for multiagent planning. In *Proceedings of the Fourth National Conference on Artificial Intelligence*, pages 121–125, August 1984.

[33] Joseph Giarratano and Gary Riley. *Expert Systems: Principles and Programming.* PWS-KENT Series in Computer Science. PWS-KENT Publishing Company, Boston, Massachusetts, 1989.

[34] G. Gottlob and A. Leitsch. On the efficiency of subsumption algorithms. *Journal of the Association for Computing Machinery*, 32(2):280–295, April 1985.

[35] Mark Green. The University of Alberta user interface management system. In *SIGGRAPH '85*, pages 205–213, July 1985.

[36] B. Hajek and G. Sasaki. Link scheduling in polynomial time. *IEEE Transactions on Information Theory*, IT-34:910–917, September 1988.

[37] Elgin Harten. Private communication.

[38] James Hendler, Austin Tate, and Mark Drummond. AI planning: Systems and techniques. *AI Magazine*, 11(2):61–77, Summer 1990.

[39] B. R. Hogencamp. GUS: A graphical user interface for capturing structural knowledge. Master's thesis, Clarkson University, Potsdam, New York, January 1987.

[40] Brian R. Hogencamp. GUS: A graphical user interface for capturing structural knowledge. Master's thesis, Clarkson University, Potsdam, New York, 13699, January 1987.

[41] Marty Humphrey. Maintaining consistent beliefs among multiple agents sharing knowledge. Master's thesis, Clarkson University, Potsdam, New York, 13699, July 1988.

[42] Robert J. K. Jacob. Using formal specifications in the desing of a human-computer interface. *Communications of the ACM*, 26(4):259–264, April 1983.

[43] A. Kershenbaum and M. J. Post. Distributed scheduling of CDMA networks with minimal information. *IEEE Transactions on Communications*, 39(1), January 1991.

[44] Kazuhiro Kuwabara and Victor R. Lesser. Extended protocol for multistage negotiation. In *Proceedings of the Ninth Workshop on Distributed Artificial Intelligence*, pages 129–161, 1989.

[45] A. L. Lansky. Behavioral specification and planning for multiagent domains. Technical Note 360, Artificial Intelligence Center, SRI International, Menlo Park, CA, November 1985.

[46] A. L. Lansky. Localized event-based reasoning for multiagent domains. Technical Note 423, Artificial Intelligence Center, SRI International, Menlo Park, CA, January 1988.

[47] A. L. Lansky. Localized search for controlling automated reasoning. Technical Report FIA-90-08-15-01, NASA Ames Research Center, Moffet Field, CA, August 1990.

[48] V. R. Lesser and D. D. Corkill. The distributed vehicle monitoring testbed: A tool for investigating distributed problem solving networks. *AI Magazine*, 4(3):15–33, Fall 1983.

[49] Victor R. Lesser and Daniel D. Corkill. Functionally-accurate, cooperative distributed systems. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-11(1):81–96, January 1981.

[50] C. Michael Lewis and Katia P. Sycara. Informed decision making in multi-specialist cooperation. In *Proceedings of the 11th International Workshop on Distributed Artificial Intelligence*, Glenn Arbor, Michigan, February 1992.

[51] P. M. Lewis, D. J. Rosenkrantz, and R. E. Stearns. Attributed translations. *Journal of Computer and System Sciences*, 9:279–307, 1974.

[52] P. M. Lewis, D. J. Rosenkrantz, and R. E. Stearns. *Compiler Design Theory.* Addison Wesley, 1976.

[53] D. J. MacIntosh and S. E. Conry. A distributed development environment for distributed expert systems. In *Proceedings of the Third Annual Expert Systems in Government Conference*, pages 19–23, Washington, D.C., October 1987. Computer Society Press of the IEEE. (also available as NAIC Technical Report TR-8714).

[54] D. J. MacIntosh and S. E. Conry. SIMULACT: A generic tool for simulating distributed systems. In *Proceedings of the Eastern Simulation Conference*, pages 18–23, Orlando, Florida, April 1987. The Society for Computer Simulation. (also available as NAIC Technical Report TR-8713).

[55] D. J. MacIntosh, S. E. Conry, and R. A. Meyer. Distributed automated reasoning: Issues in coordination, cooperation, and performance. *IEEE Transactions on Systems, Man, and Cybernetics*, 21(6), December 1991.

[56] Douglas J. MacIntosh. *Distributed Automated Reasoning: The Role of Knowledge in Distributed Problem Solving.* PhD thesis, Clarkson University, Potsdam NY, December 1989.

[57] Douglas J. Mac Intosh. *Distributed Automated Reasoning: The Role of Knowledge in Distributed Problem Solving.* PhD thesis, Clarkson University, Potsdam, New York, 13699, December 1989.

[58] Douglas J. Mac Intosh and Susan E. Conry. SIMULACT: A generic tool for simulating distributed systems. In *Proceedings of the Eastern Simulation Conference*, pages 18–23, Orlando, Florida, April 1987. The Society for Computer Simulation. (Also available as NAIC technical report TR–8713).

[59] Douglas J. Mac Intosh, Susan E. Conry, and Robert A. Meyer. Role of knowledge in distributed problem solving. In *9th Annual Distributed Artificial Intelligence Workshop*, pages 215–238. American Association for Artificial Intelligence (AAAI), September 1989. Orcas Island, Washington.

[60] Elliott Mendelson. *Introduction to Mathematical Logic.* The Wadsworth & Brooks/Cole Mathematics Series. Wadsworth & Brooks/Cole Advanced Books & Software, Monterey, California, third edition, 1987.

[61] R. A. Meyer and Susan E. Conry. Distributed artificial intelligence for communications network management. Technical Report RADC-TR-90-404, Vol IV (of 18), Rome Air Development Center, Air Force Systems Command, Griffiss Air Force Base, NY 13441-5700, 1990.

[62] Robert A. Meyer and Susan E. Conry. Distributed artificial intelligence for communications network management. Final Technical Report RADC-TR-90-404, Vol IV (of 18), Northeast Artificial Intelligence Consortium (NAIC), 1990.

[63] Daniel P. Miranker. *TREAT: A New and Efficient Match Algorithm for AI Production Systems.* Research Notes in Artificial Intelligence. Morgan Kaufmann Publishers, Inc., San Mateo, California, 1990.

[64] Dan R. Olsen Jr. Pushdown automata for user interface management. *ACM Transactions on Graphics*, 3(3):177–203, July 1984.

[65] R. P. Pope. Role recognition in multiagent distributed planning. Master's thesis, Clarkson University, Potsdam, NY, September 1988.

[66] R. P. Pope, S. E. Conry, and R. A. Meyer. DCONSA: Distributed constraint-based planning for semi-autonomous agents. In *Proceedings of the 10th International Workshop on Distributed Artificial Intelligence*, Bandera, Texas, October 1990.

[67] R. P. Pope, S. E. Conry, and R. A. Meyer. Distributing the planning process in a dynamic environment. In *Proceedings of the 11th International Workshop on Distributed Artificial Intelligence*, Glenn Arbor, Michigan, February 1992.

[68] M. J. Post, P. E. Sarachik, and A. S. Kershenbaum. A distributed evolutionary algorithm for reorganizing network communications. In *Proceedings of the IEEE Military Communications Conference (MILCOM85)*, 1985.

[69] G. A. Robinson and L. Wos. Paramodulation and theorem proving in first order theories with equality. *Machine Intelligence*, 4:135–150, 1969.

[70] J. A. Robinson. Automatic deduction with hyper-resolution. *International Journal of Computer Mathematics*, 1:227–234, 1965.

[71] J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the Association for Computing Machinery*, 12(1):23–41, January 1965.

[72] E. D. Sacerdoti. Planning in a hierarchy of abstraction spaces. *Artificial Intelligence*, 5:115–135, 1974.

[73] E. D. Sacerdoti. *A Structure for Plans and Behavior.* Elsevier-North Holland, New York, 1977.

[74] J. R. Smith. Automatic theorem proving with renamable and semantic resolution. *Journal of the Association for Computing Machinery*, 14(4):687–697, October 1967.

[75] R. G. Smith. The contract net protocol: High level communication and control in a distributed problem solver. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-10(12), December 1980.

[76] Reid G. Smith. The contract net: A formalism for the control of distributed problem solving. In *Proceedings of the Fifth International Joint Conference on Artificial Intelligence*, page 472, Los Altos, California, 1977. Morgan Kaufmann Publishers, Inc. Cambridge, Massachusetts, United States.

[77] Reid G. Smith and Randall Davis. Frameworks for cooperation in distributed problem solving. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-11(1):61–70, January 1981.

[78] Guy L. Steele Jr. *Common LISP: The Language.* Digital Press, Bedford, Massachusetts, second edition, 1990.

[79] M. J. Stefik. Planning and meta-planning. *Artificial Intelligence*, 16:141–169, 1981.

[80] M. J. Stefik. Planning with constraints. *Artificial Intelligence*, 16:111–140, 1981.

[81] K. Sycara, S. Roth, N. Sadeh, and M. Fox. Distributed constrained heuristic search. *IEEE Transactions on Systems, Man, and Cybernetics*, 21(6), December 1991.

[82] A. Tate. Project planning using a hierarchic non-linear planner. Technical Report Research Report No. 25, University of Edinburgh, Department of Artificial Intelligence, August 1976.

[83] U.S. Army Communications Command, Fort Huachuca, Arizona 85613–5300. *AN/FCC–98 Student Handout.* Training Detachment, New Equipment Training Section, U.S. Army Communications-Electronics Installation Battallion.

[84] U.S. Army Communications Command, Fort Huachuca, Arizona 85613–5300. *AN/FCC–99 Student Handout.* Training Detachment, New Equipment Training Section, U.S. Army Communications-Electronics Installation Battallion.

[85] U.S. Army Information Systems Command, Fort Huachuca, Arizona 85613–5300. *Radio Set AN/FRC–171 Student Handout.* Training Office, New Equipment Training Branch, U.S. Army Communications-Electronics Installation Battallion.

[86] S. Vere. Planning in time: Windows and durations for activities and goals. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-5(3):246–267, 1983.

[87] D. E. Wilkins. Domain-independent planning: Representation and plan generation. *Artificial Intelligence*, 22:269–301, 1984.

[88] L. Wos. The unit preference strategy in theorem proving. In *Proceedings of the AFIPS Conference*, pages 615–621. Spartan Books, 1964.

[89] L. Wos. Efficiency and completeness of the set of support strategy in theorem proving. *Journal of the Association for Computing Machinery*, 14(1):536–541, March 1965.

[90] L. Wos. The concept of demodulation in theorem proving. *Journal of the Association for Computing Machinery*, 14(1):698–709, 1967.

[91] Larry Wos, Ross Overbeek, Ewing Lusk, and Jim Boyle. *Automated Reasoning: Introduction and Applications.* Prentice-Hall, Inc, Englewood Cliffs, New Jersey, 1984.

[92] Q. Yang, D. Nau, and J. Hendler. An approach to multiple-goal planning with limited interactions. In *Working Notes AAAI Spring Symposium Series: Planning and Search*, March 1989.

[93] M. Yokoo, T. Ishida, and K. Kuwabara. Distributed constraint satisfaction for DAI problems. In *Proceedings of the 10th International Workshop on DAI*, Bandera, Texas, 1990.

*U.S. GOVERNMENT PRINTING OFFICE: 2001-610-055-10082

MISSION
OF
AFRL/INFORMATION DIRECTORATE (IF)

The advancement and application of Information Systems Science and Technology to meet Air Force unique requirements for Information Dominance and its transition to aerospace systems to meet Air Force needs.